Министерство образования и науки Российской Федерации
Ярославский государственный университет им. П. Г. Демидова

В. Л. Дольников
О. П. Якимова

Основные алгоритмы на графах

Текст лекций

Рекомендовано
Научно-методическим советом университета для студентов, обучающихся по специальности Компьютерная безопасность

Ярославль 2011

УДК 519.254
ББК В 174.2я73
Д 65

Рекомендовано
Редакционно-издательским советом университета
в качестве учебного издания. План 2011 года

Рецензенты:
Р. Н. Карасев, доктор физико-математических наук;
кафедра теории и методики преподавания информатики ЯГПУ
им. К. Д. Ушинского

Дольников, В. Л. Основные алгоритмы на графах :
Д 65 текст лекций / В. Л. Дольников, О. П. Якимова; Яросл. гос. ун-т им. П. Г. Демидова. – Ярославль : ЯрГУ, 2011. – 80 с.

ISBN 978-5-8397-0855-6

Текст лекций предназначен для студентов, обучающихся по специальности 090102.65 Компьютерная безопасность (дисциплина «Алгоритмы на графах», блок ОПД), очной формы обучения.

УДК 519.254
ББК В 174.2я73

ISBN 978-5-8397-0855-6

Введение

Теория графов – важнейший математический инструмент, широко используемый в информатике, химии, генетике, исследовании операций, лингвистике, проектировании, так как посредством графов можно описывать разнообразные реальные явления: организацию транспортных систем, сети передачи данных, человеческих взаимоотношений, структуру гена или молекулы. Возможность формального моделирования такого множества разных реальных структур позволяет программисту решать широкий круг прикладных задач.

Разработка хорошего алгоритма «с нуля» – очень трудная задача. Часто достаточно правильно построить модель задачи и применить уже известный алгоритм, решающий задачу быстро и верно.

В рамках этого пособия разбираются основные алгоритмы на графах, решающие практические задачи. Для записи алгоритма используется как естественный язык, так и язык программирования C#.

1. Начальные понятия

Даже в период становления теории графов в ней возникало немало таких задач, решение которых предполагало построение некоторых алгоритмов. Достаточно вспомнить, например, задачу о кёнигсбергских мостах, которую решил в 1736 г. Л. Эйлер. В городе Кёнигсберге (нынешний Калининград) было два острова, соединенных семью мостами с берегами реки Преголя и друг с другом так, как показано на рис. 1. Задача состояла в следующем: найти маршрут прохождения всех четырех частей суши, который начинался бы с любой из них, кончался бы на этой же части и ровно один раз проходил по каждому мосту. Исключительный вклад Л. Эйлера в решение этой задачи заключается в доказательстве, что такой маршрут построить невозможно.

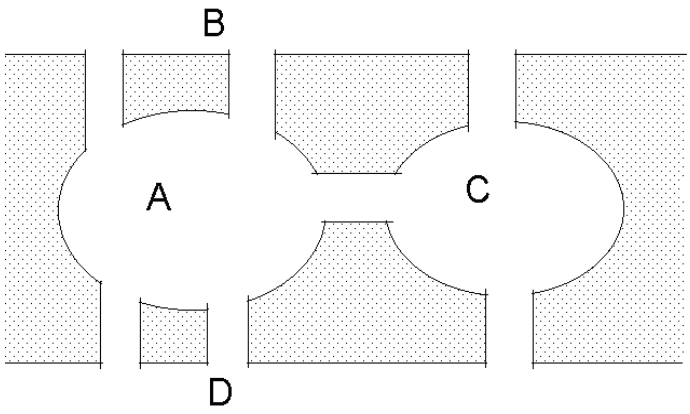


Рис. 1. Парк в г. Кёнигсберге

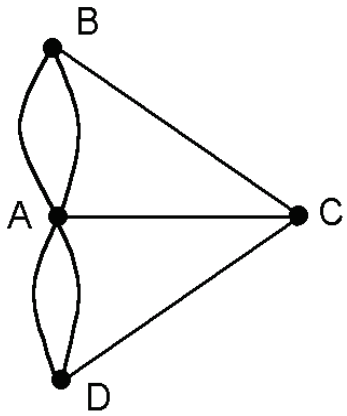


Рис. 2. Граф к задаче о мостах

Эйлер обозначил каждую часть суши точкой (вершиной), а каждый мост линией (ребром), соединяющей соответствующие точки (см. рис. 2). Полученная конструкция, состоящая из точек и соединяющих их ребер, со временем получила название граф.

Утверждение о том, что решения задачи о кенигсбергских мостах не существует, эквивалентно утверждению о невозможности обойти граф, представленный на рис. 2, проходя по каждому ребру в точности один раз. Отталкиваясь от этого частного случая, Эйлер обобщил постановку задачи и нашел критерий существования обхода у связного графа. На его основе позже был разработан алгоритм обхода связного графа.

Развитие теории графов в конце XIX и начале XX в. было связано с распространением представлений о молекулярном строении вещества и становлением теории электрических цепей. Сейчас язык и методы теории графов используются не только в математике и традиционных приложениях в химии и электротехнике, но и в информатике, социологии, лингвистике, экономике, генетике. Очевидно, что алгоритмы для работы с графами очень важны.

Внедрение вычислительной техники поставило и перед всей математикой, и перед теорией графов проблему нахождения не произвольных алгоритмов, позволяющих решать те или иные классы задач, а таких алгоритмов, которые допускали бы практическую реализацию на компьютере. Так возникла проблема практической разрешимости задач: найти эффективный или хотя бы достаточно простой в практически важных случаях алгоритм решения задачи. В этом курсе лекций будут рассмотрены наиболее яркие и ценные алгоритмы на графах.

1.1. Основные определения

Приведем несколько основных определений и понятий.

Пусть V – непустое множество, $[V]^r$ – множество всех его r-элементных подмножеств.

Графом называется пара множеств $G = (V, E)$, где $E \subseteq [V]^2$, то есть E – произвольное подмножество множества $[V]^2$.

Элементы множества V называются *вершинами* графа G, а элементы множества E – его *ребрами*.

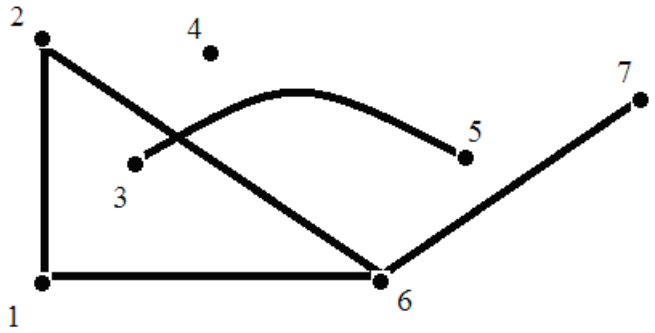


Рис. 3. Граф на множестве вершин $V = \{1, 2, 3, 4, 5, 6, 7\}$
со множеством ребер $E = \{\{1, 2\}, \{1, 6\}, \{2, 6\}, \{3, 5\}, \{6, 7\}\}$

Множество вершин графа G обозначается через $V(G)$, а множество его ребер – через $E(G)$. Число вершин графа G будем обозначать через $n(G)$, число ребер – через $m(G)$. Очевидно, что максимальное число ребер в графе на n вершинах равно $n(n\text{-}1)/2$.

Говорят, что две вершины u и v графа *смежны*, если множество $\{u, v\}$ является ребром, и не смежны в противном случае. Если $e = \{u, v\}$ – ребро, то вершины u и v называют его *концами*. В этой ситуации говорят также, что ребро e соединяет вершины u и v, и что вершина u (или v) и ребро e *инцидентны*. Два ребра называются *смежными*, если они имеют общую вершину.

Подграфом графа G = (V, E) называется граф $G_1 = (V_1,E_1)$: $V_1 \subseteq V$, $E_1 \subseteq E$, у которого все вершины и ребра принадлежат G.

Исключительную роль во многих прикладных задачах играет остовный подграф графа.

Остовный подграф – это подграф графа G, содержащий все его вершины.

Подграф G_0 на подмножестве вершин $V_0 \subseteq V$ называется порожденным, если множество его ребер совпадает с множеством всех ребер графа G, оба конца которых принадлежат V_0.

Графы удобно изображать в виде рисунков, состоящих из точек и линий, соединяющих некоторые из этих точек. При этом точки соответствуют вершинам графа, а линии, соединяющие пары точек, – ребрам. На рис. 4 изображен граф G и два его подграфа G_1 и G_2. G_1 – порожденный подграф G, а G_2 – нет. G_2 – остовный подграф G, а G_1 нет. Вершины 1 и 2 графа G смежные, а 1 и 3 нет. Также смежными являются ребра {1, 4} и {4, 5}. Вершина 1 инцидентна ребрам {1, 4} и {1, 2}.

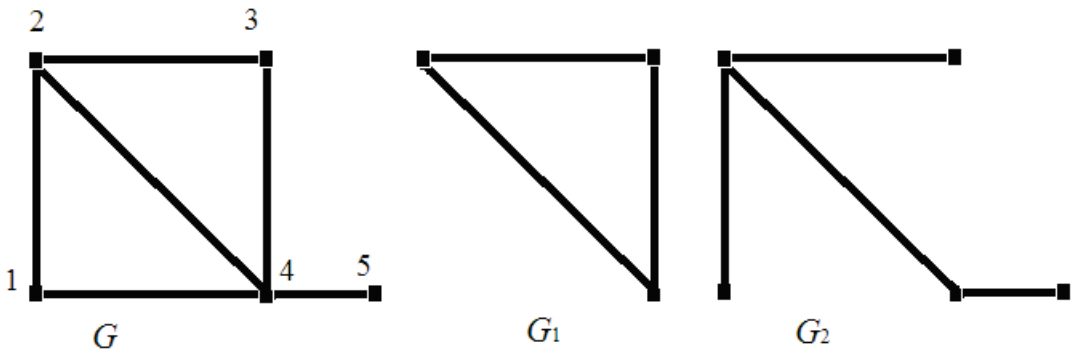


Рис. 4.

При решении некоторых практических задач приведенное выше определение графа оказывается недостаточным и приходится рассматривать более общие объекты. Многие задачи анализа программ, возникающие при оптимизации, трансляции, проверке правильности, тестировании и т. д., значительно упрощаются, если их рассматривать на теоретико-графовых моделях. В основе моделей лежит управляющий граф программы. Каждый оператор языка программирования изображается отдельной вершиной. Две вершины смежны, если между соответствующими операторами есть передача управления. Точнее, оператор, после которого производится передача управления, представляется началом дуги; оператор, на который передается управление, – концом дуги, каждая передача управления – дугой. Такие графы называются *ориентированными*.

Ориентированный граф состоит из конечного непустого мно-жества V вершин и заданного набора X^2 упорядоченных пар различных вершин. Элементы из X называются ориентированными ребрами или дугами.

Изучаются также мультиграфы.

Мультиграфом называется граф, в котором пары вершин могут соединяться более чем одним ребром; эти ребра называются кратными.

Если в мультиграфе ребра имеют ориентацию, то подобный граф называется ориентированным мультиграфом.

Дальнейшее обобщение состоит в том, что, кроме кратных ребер, допускаются еще и *петли*, то есть ребра, соединяющие вершину саму с собой. Такой граф называется *псевдографом*.

На рис. 5 представлены описанные виды графов: G_1 – мультиграф, G_2 – псевдограф, G_3 – орграф. Следует отметить, что в литературе не существует единства терминологии, и поэтому надо быть внимательным к определениям при чтении различных источников.

Далее, как правило, под термином «граф» мы будем понимать именно *обыкновенный граф*, то есть конечный неориентированный граф без петель и кратных ребер.

Приведем примеры некоторых графов специального вида. Важную роль в теории графов имеет понятие полного графа, или клики.

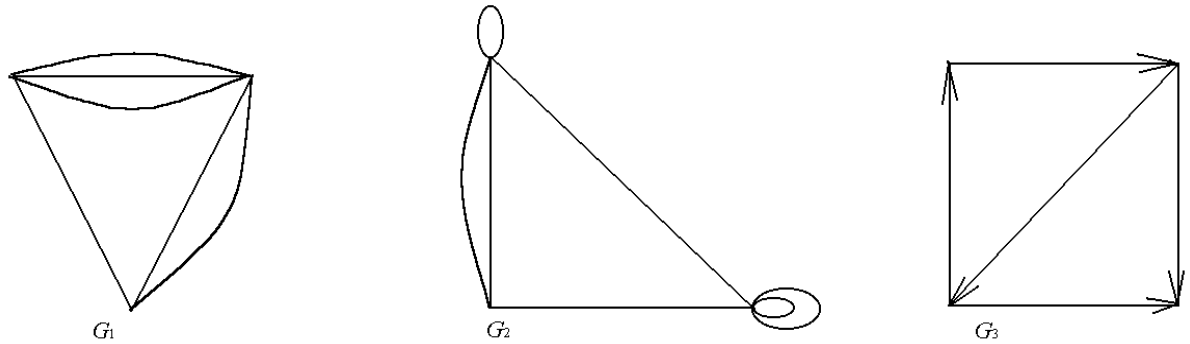


Рис. 5. Мультиграф, псевдограф и орграф

Граф G называется полным (кликой), если любые две его вершины смежны.

Полный граф на n вершинах обозначается K_n. Для числа ребер полного графа выполняется равенство $m(K_n) = C_n^2 = \frac{n(n-1)}{2}$.

Противоположным к понятию полного графа является понятие пустого графа.

Граф называется пустым, если в нем нет ребер. Пустой граф на n вершинах обозначается O_n.

На рис. 6 приведены изображения полного и пустого графа на четырех вершинах.

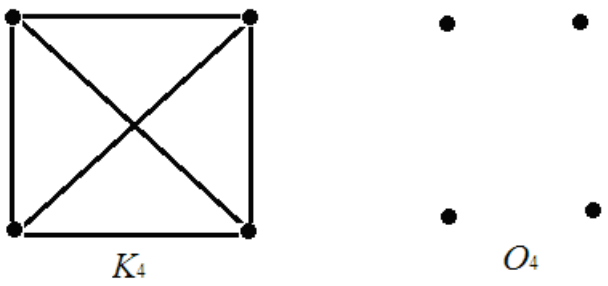


Рис. 6.

На рис. 7 показаны *простые циклы* C_n (n = 3, 4) и *простые пути* P_n (n = 2, 3, 4). Очевидно, что $K_2 = P_2$, а $K_3 = C_3$.

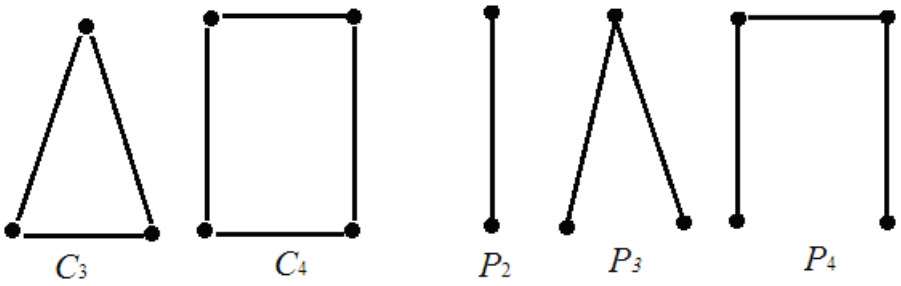


Рис. 7. Простые циклы и простые пути

В теории графов наиболее изученными являются двудольные графы, к которым сводятся многие практические задачи, например задача о назначениях.

Граф называется двудольным, если существует такое разбиение множества его вершин на две части (доли), что концы каждого ребра принадлежат разным частям.

Если при этом любые две вершины, входящие в разные доли, смежны, то граф называется *полным двудольным*. Полный двудольный граф, доли которого состоят из p и из q вершин, обозначается символом $K_{p,q}$. На рис. 8 изображен граф $K_{3,3}$.

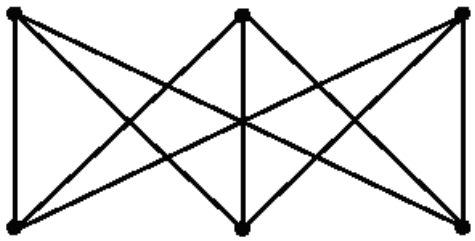


Рис. 8. Полный двудольный граф $K_{3,3}$

1.2. Представление графа в памяти компьютера

Выбор соответствующей структуры данных для представления графа имеет принципиальное значение при разработке эффективных алгоритмов. При решении задач используются следующие два стандартных способа представления графа $G = (V, E)$: как набора списков смежных вершин или как матрицы смежности. Оба способа представления применимы как для ориентированных, так и для неориентированных графов. Обычно более предпочтительно представление с помощью списков смежности, поскольку оно обеспечивает компактное представление разре-

женных графов, т. е. таких, для которых число ребер гораздо меньше квадрата числа вершин.

Представление при помощи матрицы смежности предпочтительнее в случае плотных графов, т. е. когда число ребер близко к $n(G)^2$ или когда нам надо иметь возможность быстро определить, имеется ли ребро, соединяющие две данные вершины. Например, алгоритм поиска кратчайшего пути для всех пар вершин использует представление графов именно в виде матриц смежности.

Представление графа $G = (V, E)$ в виде списка смежности использует массив *Sp_reber* из $n(G)$ списков, по одному для каждой вершины из V. Для каждой вершины $u \in V$ список *Sp_reber*[u] содержит все вершины v, такие что $\{u, v\} \in E$, т. е. *Sp_reber*[u] состоит из всех вершин, смежных с u в графе G (список может содержать и не сами вершины, а указатели на них). Вершины в каждом списке обычно хранятся в произвольном порядке.

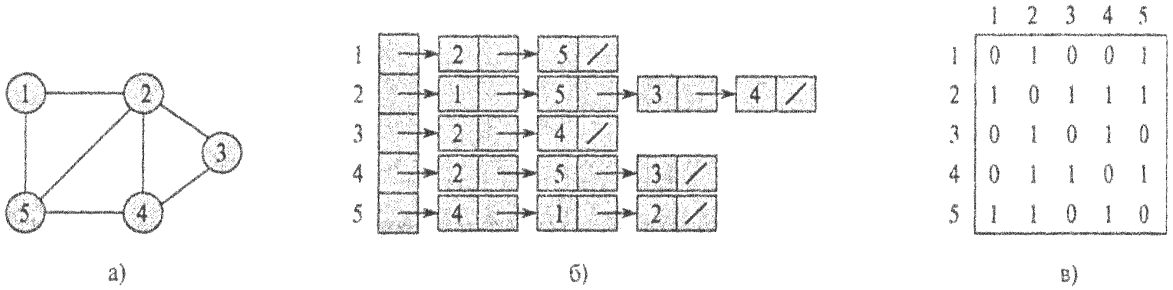


Рис. 9. Два представления неориентированного графа

На рис. 9 показано представление графа в виде списка смежности и в виде матрицы смежности. Аналогично, на рис. 10 показаны список и матрица смежности ориентированного графа.

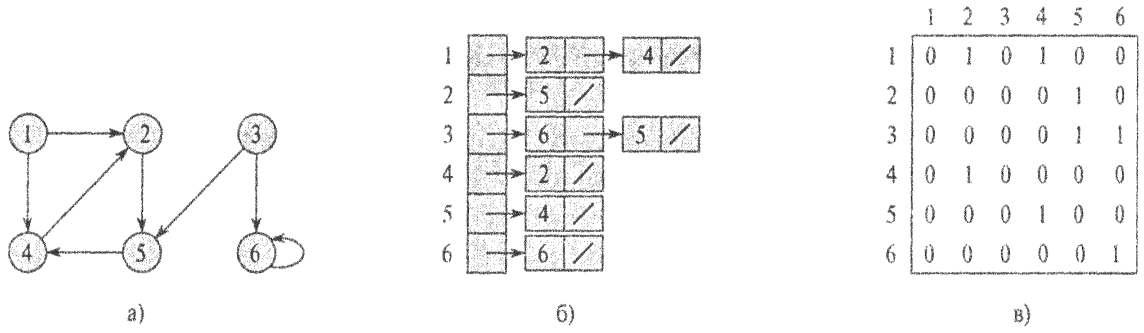


Рис. 10. Два представления ориентированного графа

Если G – ориентированный граф, то сумма длин всех списков смежности равна $m(G)$, поскольку ребру $\{u, v\}$ однозначно соответствует элемент v в списке *Sp_reber* [u]. Если G – неориентированный граф, то сумма длин всех списков смежности равна $2m(G)$, поскольку ребро $\{u,v\}$, будучи неориентированным, появляется в списке *Sp_reber* [v] как u и в списке *Sp_reber* [u] – как v. Как для ориентированных, так и для неориентированных графов представление в виде списков требует объем памяти, равный $O(n + m)$.

Представление графа $G = (V, E)$ с помощью матрицы смежности предполагает, что вершины перенумерованы в некотором порядке числами 1, 2, ..., n. В таком случае представление графа G с использованием матрицы смежности представляет собой матрицу A размером $n \times n$, такую что

$$a_{ij} = \begin{cases} 1, & \text{если } \{i,j\} \in E \\ 0, & \text{в противном случае.} \end{cases}$$

На рис. 9 в) и 10 в) показаны представления с использованием матрицы смежности неориентированного и ориентированного графов, показанных на рис. 9 а) и 10 а) соответственно. Матрица смежности графа требует объем памяти, равный n^2, независимо от количества ребер графа. Ниже приведен листинг, иллюстрирующий реализацию графа в виде матрицы смежности.

```
public  class graph
{
public int[,] matr_smeznosti;
public int kol_vershn; // количество вершин графа
// конструктор, считывающий граф из файла
public graph(string fileName)
{
int n, j;
string line;
char[] delimiterChars = new char[] { ',', ' ', ';', '.' };
using (StreamReader file = new StreamReader(fileName))
{
```

```
        n = int.Parse(file.ReadLine());  // ввод размерности
        this.kol_vershn = n;
        this.matr_smeznosti = new int[n, n];
        // ввод матрицы
        for (int i = 0; (i < n) && ((line = file.ReadLine()) != null); i++)
        {
            string[] numbers = line.Split(delimiterChars,
            StringSplitOptions.RemoveEmptyEntries);
            j = 0;
            foreach (string numString in numbers)
            {
                int x;
                bool canConvert = int.TryParse(numString, out x);
                if (canConvert == true)
                {
                    this.matr_smeznosti[i, j] = x;
                    j++;
                }
            }
        }
    }
} }
```

Листинг 1. Реализация графа в виде матрицы смежности

Списки и матрица смежности легко адаптируются для представления взвешенных графов, т. е. графов, с каждым ребром которых связан определенный вес, обычно определяемый весовой функцией w: $E \rightarrow R$. В этом случае, в каждый элемент списка добавляется весовое поле, а для матрицы смежности

$$a_{ij} = \begin{cases} w_{ij}, \text{если } \{i, j\} \in E \\ 0, \text{в противном случае.} \end{cases}$$

Потенциальный недостаток представления при помощи списков смежности заключается в том, что при этом нет более быстрого способа определить, имеется ли данное ребро (u, v) в

графе, чем поиск *v* в списке *Sp_reber* [*u*]. Этот недостаток можно устранить ценой использования асимптотически большего количества памяти и представления графа с помощью матрицы смежности. Следует отметить, что простота последней делает ее предпочтительной при работе с небольшими графами.

Далее при представлении графов будем использовать оба способа в зависимости от решаемой задачи. Ниже приведен листинг, иллюстрирующий реализацию графа в виде списка смежности.

```
internal class rebro // внутренний класс ребро
{
public int v_na { get; set; }  // смежная вершина
public int ves   { get; set; }   // вес ребра( если есть)
}
public class graph_edgeList
{
   internal int kol_v_n { get; set; } // количество вершин графа
   internal List<rebro>[] sp_reber;//массив списков смежных в-н
// конструктор, считывающий граф из файла
   public graph_edgeList(string filename)
   {
char[] separator = new char[] { '.', ' ', ',', ';' };
using (StreamReader f = new StreamReader(filename))
{
  int n = int.Parse(f.ReadLine()); // количество вершин
          sp_reber = new List<rebro>[n];
  string line = f.ReadLine();
  for (int i = 0; (i < sp_reber.Length) && (line != null); i++)
  {
       List<rebro> rlist = new List<rebro>();
```

```
            string[] values = line.Split(separator,
            StringSplitOptions.RemoveEmptyEntries);
            if (values.Length > 0)
            {
               for (int j = 0; j < values.Length; j++)
               {
                  int v = int.Parse(values[j]);
                  //int w = int.Parse(values[j + 1]);
                  rebro r = new rebro { v_na = v};
                  rlist.Add(r);
               }
            }
            line = f.ReadLine();
            sp_reber[i] = rlist;
         }
         kol_v_n = n;
      }
   } }
```

Листинг 2. Реализация графа в виде списка смежности

1.3 Анализ алгоритмов

Неформальное определение алгоритма может быть следующим: алгоритм – это любая корректно определенная вычислительная процедура, на вход которой подается некоторая величина или набор величин и результатом выполнения которой является выходная величина или набор значений. Таким образом, алгоритм представляет собой последовательность вычислительных шагов, преобразующих входные величины в выходные.

Алгоритмы – важнейшая часть информатики, при их изучении можно обойтись без компьютера или знания языка программирования. Но требуется как-то сравнивать алгоритмы, какой из них лучше, эффективнее. При оценке алгоритмов считается, что они выполняются на некоторой гипотетической машине, где для

исполнения любой простой операции (+, *, –, =, if, call) требуется ровно один временной шаг, циклы и подпрограммы не считаются простыми операциями. Время выполнения цикла равно количеству его итераций, подпрограммы – количеству простых операций внутри. Исходные данные для решения любой задачи размещаются в памяти машины, будем называть их входом задачи. Размером (или длиною) входа считается число ячеек памяти компьютера, занятых входом. Считается, что память гипотетической машины не ограничена, а время доступа занимает один временной шаг.

С помощью этой модели можно подсчитать для любого алгоритма число шагов, которые он затратит на решение экземпляра задачи. Заметим, что число шагов зависит от длины входных данных n. При анализе алгоритма любой задачи нас в первую очередь будет интересовать зависимость времени его работы от размера входа. Однако это время зависит не только от размера входа, но и от других параметров задачи.

Например, на входе задачи n натуральных чисел $a_1, a_2, ..., a_n$ и требуется найти среди них число, кратное трем. Очевидный алгоритм решения состоит в следующем: будем проверять поочередно делимость элементов исходного списка данных на три, пока не встретится число a_i, кратное трем. В зависимости от расположения такого элемента a_i в списке исходных данных потребуется от 1 до n проверок на делимость. Наихудшим будет случай, когда подходящего числа в списке нет или оно расположено последним. При определении понятия *трудоемкость алгоритма* будем ориентироваться на такого рода наихудший случай.

Трудоемкость (*временная сложность*) алгоритма решения задачи – это функция f, которая ставит в соответствие каждому натуральному числу n максимальное число шагов (время работы) алгоритма $f(n)$ на входных данных длины n.

Напрямую работать с функцией трудоемкости достаточно трудно, так как часто она требует слишком много информации для точного определения. Например, для некоторого алгоритма функция имеет вид: $f'(n) = 548n^2 + 3792n + 62\log_2 n + 2801$. Точный анализ такой функции достаточно труден. Поэтому на практике пользуются асимптотическими обозначениями.

Говорят, что неотрицательная функция $f(n)$ не превосходит по порядку функцию $g(n)$, если существует такая константа $c > 0$, что $f(n) \le cg(n)$ для всех $n \ge n_0$, при этом пишут $f(n) = O(g(n))$.

Алгоритм с трудоемкостью $O(n)$, где n – длина входа, называют *линейным*. Линейный алгоритм ограниченное константой число раз просматривает входную информацию и по сути является неулучшаемым (по порядку) в смысле трудоемкости.

Алгоритм с трудоемкостью $O(n^c)$, где c – константа, называется *полиномиальным*. Согласно общепринятой точке зрения алгоритм является эффективным, если он имеет полиномиальную сложность.

Прежде всего в этом пособии будут рассматриваться полиномиальные алгоритмы.

Упражнения

1. Ориентированный граф хранится в виде списков смежности. Сколько операций нужно, чтобы вычислить исходящие степени всех вершин графа? А входящие степени?

2. Укажите представление с использованием списков смежности и матрицы смежности для графа $K_{3,3}$.

3. Транспонированием орграфа назовем операцию, при которой направление каждого ребра меняется на противоположное. Опишите эффективный алгоритм транспонирования орграфа как для представления с использованием списков смежности, так и для матриц смежности. Проанализируйте время работы ваших алгоритмов.

4. Мультиграф $G = (V, E)$ хранится в виде списков смежных вершин. Опишите алгоритм со временем работы $O(V + E)$ для преобразования его в обычный граф $G' = (V', E')$, при этом кратные ребра заменены обычными и удалены ребра-циклы.

5. Составьте таблицу, в которой вычисляется время работы алгоритмов с трудоемкостью $\log(n)$, n, $n\log(n)$, $n2$, $n!$ на входных данных длиной $n = 100, 1000, 10^4, 10^6, 10^8, 10^{10}$.

6. Предположим, что время исполнения алгоритма в наихудшем случае определяется как $O(n^2)$. Возможно ли, что для некоторых входных данных это время будет $O(n)$?

2. Алгоритмы обхода графа

Все основные служебные операции при работе с графами (например, преобразование графа из одного представления в другое, распечатка или рисование графа) предполагают его систематический обход, то есть посещение каждой вершины и каждого ребра графа. Если представить лабиринт в виде графа, где ребра – проходы, а вершины – точки пересечения проходов, то любой правильный алгоритм обхода графа должен найти выход из произвольного лабиринта. Наиболее известными из таких алгоритмов являются *поиск в глубину* (depth first search, DFS) и *поиск в ширину* (breadth-first search, BFS), причем они являются базисными для многих других алгоритмов решения прикладных задач, включающих обход графа, что и будет показано в дальнейшем.

Ключевая идея обхода графа – пометить каждую вершину при первом ее посещении и хранить информацию о тех вершинах, не все ребра из которых просмотрены. В мифах Древней Греции Тесей использовал клубок ниток, который ему дала Ариадна, для обхода лабиринта, мальчик-с-пальчик помечал пройденный путь камешками или крошками, для обхода графа будут применяться перечислимые типы. В процессе обхода графа каждая его вершина будет находиться в одном из трех состояний:

- *неоткрытая* – первоначальное состояние вершины;
- *открытая* – вершина обнаружена, но инцидентные ей ребра не просмотрены;
- *обработанная* (*помеченная*) – все ребра, инцидентные этой вершине, посещены.

Понятно, что каждая вершина графа последовательно принимает все эти состояния. Первоначально открытой является только одна вершина – начало обхода графа.

Замечание: предложенные методы обходят все вершины, содержащиеся в одной компонете связности с начальной вершиной. Поэтому можно использовать любой из алгоритмов обхода для определения связности графа.

2.1. Поиск в ширину

Пусть задан граф $G = (V, E)$ и выделена *начальная* вершина v. Алгоритм поиска в ширину систематически обходит все ребра графа G для «открытия» всех вершин, достижимых из v. В процессе обхода строится дерево поиска в ширину с корнем в начальной вершине, содержащее все достижимые вершины. Заметим, что расстояние (количество ребер) от корневой вершины до любой вершины этого дерева является кратчайшим.

Поиск в ширину имеет такое название, так как в процессе обхода графа помечаются все вершины на расстоянии k, прежде чем начнется обработка любой вершины на расстоянии $k + 1$.

Алгоритм работает как для ориентированных, так и для неориентированных графов.

Идея алгоритма: все вершины, смежные с начальной, открываются, то есть помещаются в список, и получают единичную пометку. После этого начальная вершина обработана полностью и имеет пометку 2.

Следующей текущей вершиной становится первая вершина списка. Все ранее не помеченные вершины, смежные с текущей, заносятся в конец списка (становятся открытыми). Текущая вершина удаляется из списка и помечается числом 2. Процесс продолжается, пока список вершин не пуст. Такая организация списка данных называется очередью.

Рассмотрим пример (рис. 11). Исходный граф на левом рисунке. На правом рисунке рядом с вершинами в скобках указана очередность просмотра вершин графа. Ребра, образующие дерево поиска в ширину, выделены жирным.

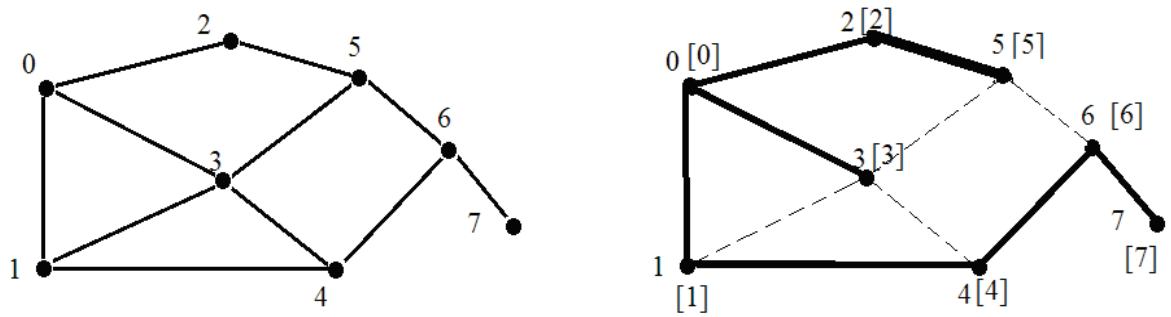


Рис. 11. Просмотр вершин графа в процессе поиска в ширину

При реализации поиска в ширину используется массив пометок *Mark,* массив предков *Parent* и очередь *Q*. Первоначально каждой вершине в массиве *Mark* соответствует значение 0, то есть вершина неоткрытая. Вершина открывается при первом посещении, и ее пометка изменяется на 1. Когда все ребра, исходящие из вершины, исследованы, то она считается обработанной и имеет пометку 2.

При просмотре списка вершин, смежных с текущей, открываются новые вершины. При этом их предком считается текущая вершина. Эта информация сохраняется в массиве *Parent* и позволяет восстановить дерево поиска в ширину. Для графа, изображенного на рис. 11, массив предков будет иметь вид:

Вершина	0	1	2	3	4	5	6	7
Предок	0	0	0	0	1	2	4	6

Граф задается матрицей смежности. Код алгоритма приведен на листинге 3.

```
public class BreadthFirstSearchAlgm
{ // Алгоритм обхода графа «Поиск в ширину»
    public void BFS(graph g)
    {
        int[] Mark = new int[g.kol_vershn]; // массив пометок
        int[] Parent = new int[g.kol_vershn]; // массив предков
        for (int i = 0; i < g.kol_vershn; i++)
        {
            Mark[i] = 0;
            Parent[i] = 0;
        }
        Console.WriteLine("Вершины в порядке обхода");
        Queue<int> Q = new Queue<int>(); // создание очереди
        int v = 0;          // задание начальной вершины
        Mark[v] = 1;     // пометим нач. вершину
        Q.Enqueue(v);   // поместим нач. вершину в очередь
        Console.Write("{0}  ", v);
```

```
        while (Q.Count != 0) //Пока очередь не исчерпана
        { //взять из очереди очередную вершину
              v = Q.Dequeue();
              for (int i = 0; i < g.kol_vershn; i++)
              {
                  if ((g.matr_smeznosti[v, i] != 0) && (Mark[i] == 0))
                  {                       // все непомеченные вершины,
                     Mark[i] = 1;    // смежные с текущей, помечаются
                     Q.Enqueue(i);  // и помещаются в конец очереди
                     Parent[i] =v;    // v  – предок открытой вершины
                     Console.Write("{0}  ", i);
                  }
              }
              Mark[v] = 2; // вершина обработана
        }
Console.WriteLine();
} }
```

Листинг 3. Обход графа в ширину

Трудоемкость данного алгоритма $O(n^2)$, где n – количество вершин графа. Действительно, каждая вершина открывается и помещается в очередь только один раз, поэтому цикл по вершинам очереди выполняется не больше n раз. В цикл while вложен цикл по всем вершинам графа, который также выполняется n раз. Если использовать представление графа в виде списков смежности, то трудоемкость была бы $O(n+m)$, где m – количество ребер.

2.2. Применение поиска в ширину

Массив *Parent*, который строится в процессе обхода графа, очень полезен для поиска кратчайших путей в графе. У всех вершин, кроме начальной, есть предок. Отношение предшествования формирует дерево поиска в ширину с корнем в начальной вершине. Вершины добавляются в очередь в порядке возрастающего расстояния (в смысле количества ребер на пути от корневой вершины до данной), поэтому дерево поиска определяет крат-

чайший путь от начальной вершины до любой другой вершины $u \in V$. Этот путь можно воссоздать, следуя по цепи предшественников от u к корню, то есть фактически в обратном направлении. Так как это направление противоположно тому, в котором фактически выполнялся проход, то после восстановления пути его нужно обратить или использовать рекурсивный метод, как показано на листинге ниже.

```
public void PrintPath(int v_begin, int v_end)
 {
        int u = Parent[v_end];
        if (u == v_begin)
        {
           Console.Write(" {0}  {1} ", u, v_end);
           return;
         }
         else if (Parent[v_end] == 0)
         {
           Console.WriteLine(" Пути из {0} в {1} нет", v_begin, v_end);
         }
         else PrintPath(v_begin, u);
         Console.Write("{0}  ", v_end);
}
```

Листинг 4. Метод печати пути по дереву поиска в ширину

Другим применением поиска в ширину является выделение его компонент связности. Граф называется связным, если существует путь между любыми двумя его вершинами. Например, все компьютеры, включенные в сеть Интернет, образуют связный граф, и хотя отдельная пара компьютеров может быть не соединена напрямую, но от каждого компьютера можно передать информацию к любому другому (есть путь из любой вершины графа в любую другую). Компонентой связности неориентиро-

ванного графа называется его связный подграф, максимальный по включению вершин.

Для выделения компонент связности графа поиск в ширину запускается из произволной вершины. После окончания работы алгоритма проводится анализ массива пометок *Mark*. Если все вершины помечены, то граф состоит из одной компоненты связности, то есть является связным. Если нет, то алгоритм запускается из произвольной непомеченной вершины. При необходимости процесс повторяется.

Также с помощью поиска в ширину легко можно определить, является ли граф двудольным, содержит или нет циклы нечетной длины. Модификация алгоритма для решения этих задач оставляется читателю в качестве упражнения.

2.3. Поиск в глубину

Основная идея поиска в глубину, как и следует из его названия, состоит в том, чтобы идти «вглубь» графа, пока это возможно. Просмотр начинается с некоторой начальной вершины v. Она считается «открытой». Выбирается вершина u, смежная с v, открывается, то есть помечается значением 1, и процесс повторяется с одной из вершин, смежных с u.

Если на очередном шаге мы работаем с вершиной q и нет вершин, смежных с q, и неоткрытых, то возвращаемся на предыдущий шаг. Если попадаем в начальную вершину, то обход графа завершен.

В процессе обхода графа строится дерево поиска в глубину. Вершина v, через которую открывается вершина u, является ее предком. Эта информация сохраняется в массиве *Parent*.

Для определения состояния вершины (неоткрытая, открытая или обработанная), как и раньше, используется массив пометок *Mark*. Если вершина i открыта, то $Mark[i] = 1$, обработана – 2, в исходном состоянии $Mark[i] = 0$. Для хранения последовательности открытых, но необработанных вершин используется стек. Эта структура данных обеспечивает возврат к предыдущей открытой вершине.

Также с каждой вершиной связываются две метки времени: время открытия вершины *time_in* и время завершения ее обработки *time_out,* которые обладают рядом интересных свойств.

Рассмотрим их на примере графа, изображенного на рис. 12 слева. Пусть поиск в глубину начинается с нулевой вершины. На рис. 12 справа у вершин в скобках указаны две метки времени. Фактически первая метка – это та очередность, в которой вершины графа просматривались в процессе обхода. Разница во времени обработки и открытия для вершины *v* дает информацию о количестве потомков этой вершины в дереве поиска в глубину. Показания часов увеличиваются на единицу при каждом входе и каждом выходе из вершины, поэтому количество потомков данной вершины будет равно половине разности между моментом завершения обработки вершины и моментом ее открытия.

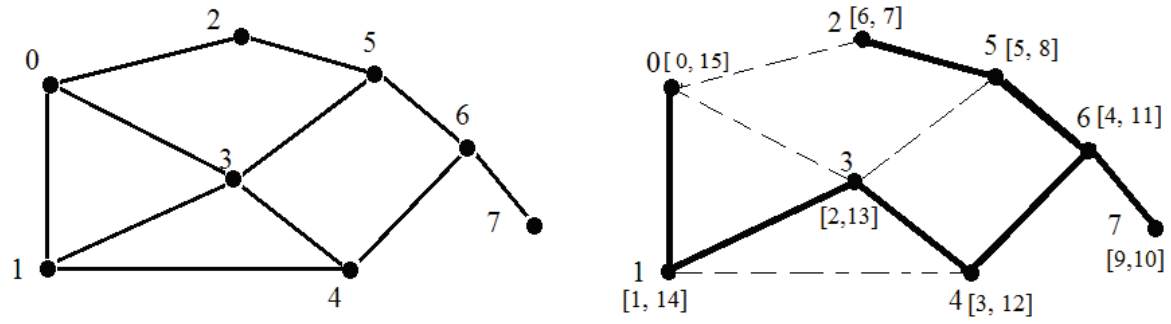


Рис. 12. Порядок просмотра вершин графа при поиске в глубину

При реализации алгоритма был использован рекурсивный метод. Код приведен на листинге 5.

```
public class DepthFirstSearchAlgm
 { // алгоритм обхода графа "сначала вглубь"
      int[] Mark;     // массив пометок
      int[] Parent;   // массив предков
      int[] time_in;  // время открытия каждой вершины
      int[] time_out; // время завершения обработки
      int time;       // переменная отсчета времени

    public void  DFS(graph g)
    {// инициализация переменных
        Mark = new int[g.kol_vershn];
```

```
        Parent = new int[g.kol_vershn];
        time_in = new int[g.kol_vershn];
        time_out = new int[g.kol_vershn];
        time = 0;
        for (int i = 0; i < g.kol_vershn; i++)
        {
            Mark[i] = 0;
            Parent[i] = 0;
            time_in[i] = 0;
            time_out[i] = 0;
        }
        Console.WriteLine("Вершины в порядке обхода");
        DFS_traversal(g, 0);
        Console.WriteLine();
    }
    void DFS_traversal(graph g, int v)
{
// рекурсивный метод обхода
        Mark[v] = 1;  // v – открыта
        time_in[v] = time; // время открытия вершины
        time++;
        Console.Write("{0}  ", v);
        for (int i = 0; i < g.kol_vershn; i++)
         {// если вершины смежны  и в i еще не были
            if ((g.matr_smeznosti[v, i] != 0)&& (Mark[i] == 0))
            {
                Parent[i] = v;  // v – предок для i
                DFS_traversal(g, i); //рекурсивный вызов
            }
          }
        Mark[v] = 2; // вершина обработана
```

```
        time_out[v] = time; //время обработки вершины
        time++;
    }
}
```

Листинг 5. Обход графа в глубину

Оценим трудоемкость алгоритма. Метод *DFS_traversal* вызывается ровно один раз для каждой вершины $v \in V$, так как он вызывается только для неоткрытых вершин, и первое, что он делает, – это помечает переданную в качестве параметра вершину единицей. Но для каждой вершины осуществляется просмотр и проверка на смежность всех остальных вершин. Итого $O(n^2)$, где n – количество вершин в графе. Если использовать представление графа в виде списков смежности, то трудоемкость была бы $O(n + m)$, где m – количество ребер.

2.4. Применение обхода в глубину

Важным свойством обхода в глубину является то, что он разбивает ребра неориентированного графа на два класса: *древесные* (ребра дерева поиска в глубину) и *обратные*. Древесные ребра можно восстановить, используя отношение предок-потомок, сохраненное в массив *Parent*. Обратные ребра – это те ребра, которые идут от потомка к одному из предков. Все ребра неориентированного графа попадают в одну из этих категорий. На рис. 12 справа древесные ребра нарисованы сплошной линией, а обратные пунктиром.

Если в неориентированном графе нет обратных ребер, то, значит, все ребра древесные. Следовательно, это – граф без циклов (ациклический). Модификация алгоритма так, чтобы для каждого ребра печатался его тип, предоставляется читателю в качестве упражнения.

Поиск в глубину также применяется при решении следующей задачи: имеется компьютерная сеть, состоящая из центров хранения и переработки информации. Некоторые пары центров соединены каналами. Обмен информацией между любыми двумя центрами осуществляется либо непосредственно по соединяю-

щему их каналу, если он есть, либо через другие каналы и центры. Сеть считается исправной, если каждая пара центров в состоянии обмениваться информацией. Такой сети естественно сопоставить граф: вершины – центры, ребра – каналы сети. Тогда исправной сети будет соответствовать связный граф.

Важным понятием является надежность (живучесть) сети, под которой обычно подразумевают способность сети функционировать при выходе из строя одного или нескольких центров или (и) каналов. Ясно, что менее надежной следует считать ту сеть, исправность которой нарушается при повреждении меньшего количества элементов. Оказывается, надежность сети можно измерять на основе вводимых ниже определений.

Вершина v графа G называется *точкой сочленения* (или *разделяющей* вершиной), если граф $G \setminus v$ имеет больше компонент связности, чем G. В частности, если G связен и v – точка сочленения, то $G \setminus v$ не связен. Аналогично, ребро графа называется *мостом*, если его удаление увеличивает число компонент. Таким образом, точки сочленения и мосты – это своего рода «узкие места» графа. Понятно, что концевая вершина моста является точкой сочленения, если в графе есть другие ребра, инцидентные этой вершине. У графа G, изображенного на рисунке 13, вершины 2, 3, 5, 6 – точки сочленения, ребро {3, 5} является мостом.

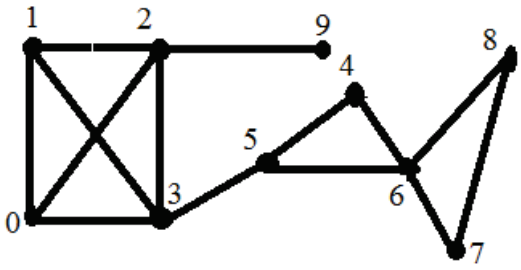


Рис. 13. Граф G

Возвращаясь к рассмотренной выше сети, нетрудно заметить, что мостам и точкам сочленения отвечают наиболее уязвимые места сети. Для их определения используется модифицированный алгоритм поиска в глубину.

Как уже говорилось выше, алгоритм поиска в глубину в процессе работы строит дерево. В дереве поиска в глубину ребро

{*v*, *w*} является мостом тогда и только тогда, когда не существует обратное ребро, которое соединяет одного из потомков *w* с каким-нибудь предком *v*. Действительно, если такое ребро существует, то {*v*, *w*} не может быть мостом. С другой стороны, если {*v*, *w*} не мост, то в графе должен быть другой путь из *v* в *w*, отличный от этого ребра. Понятно, что каждый подобный путь должен содержать одно из обратных ребер от потомков *w* к предкам *v*.

Для определения точек сочленения будем использовать очередность просмотра вершин в процессе поиска в глубину, которая записывается в массиве *time_in*. Если ребро {*v*, *u*} – обратное, и *v* не предок *u*, то информацию о том, что *time_in*[*v*] больше *time_in*[*u*], можно использовать для пометки вершин *v* и *u* как вершин, принадлежащих одной компоненте двусвязности. Введем массив *LowDFS* и будем помечать вершины графа, принадлежащие одной компоненте двусвязности, одним значением метки в этом массиве. Первоначальное значение метки совпадает со значением соответствующего элемента массива *time_in*. При нахождении обратного ребра {*v*, *u*} будем изменять значение метки вершины *v*: *LowDFS*[*v*] = *Min*(*LowDFS*[*v*], *time_in*[*u*]), так как эти вершины из одной компоненты двусвязности. Кроме того, необходимо добавить смену значения метки у вершины *v* ребра {*v*, *u*} на выходе из просмотра в глубину в том случае, если значение метки вершины *u* меньше, чем метка вершины *v* (*LowDFS*[*v*] = *Min(LowDFS*[*v*], *LowDFS*[*u*])).

Если для некоторой вершины *u* значение в массиве *LowDFS*[*u*] не меньше времени открытия этой вершины, то она является точкой сочленения. Также начальная вершина тоже будет точкой сочленения, если у нее не меньше двух независимых потомков.

Модифицированный код поиска в глубину приведен в листинге 6.

```
void DFS_traversal(graph g, int tek_vna)
 {// рекурсивный метод обхода
    int u;
    int kids = 0;       // количество потомков
```

```
bool art = false; // флаг точки сочленения
Mark[tek_vna] = 1;
time++;
LowDFS[tek_vna] = time;
time_in[tek_vna] = time;  // время открытия вершины

for (u = 0; u < g.kol_vershn; u++)
{// если вершины смежны
   if (g.matr_smeznosti[tek_vna, u] != 0)
   {
      if( u == Parent[tek_vna]) continue;
      if(Mark[u] == 0) // если вершина u неоткрытая
      {
       Parent[u] = tek_vna;  // tek_vna – предок для u
       DFS_traversal(g, u);    //рекурсивный вызов
       LowDFS[tek_vna] = Math.Min(LowDFS[u], LowDFS[tek_vna]);
          if (LowDFS[u] >= time_in[tek_vna])
          {
                art = true;
           }
           kids++;
           if (LowDFS[u] == time_in[u])
           {
               Console.WriteLine("{0}  {1}- мост ", tek_vna, u);
            }
        }
      else  // вершина u  – открытая
     {  // корректируем значения массива LowDFS
       LowDFS[tek_vna] = Math.Min(LowDFS[tek_vna],time_in[u]);
      }
    }
```

```
        }
        Mark[tek_vna] = 2; // вершина обработана
        if (Parent[tek_vna] == -1)
            art = (kids >= 2);
        if (art)
            Console.WriteLine(" v = {0} – точка сочленения ", tek_vna);
}
```

Листинг 6. Поиск мостов и точек сочленения в графе

Похожий алгоритм для ориентированных графов принадлежит Тарьяну. Подробнее о нем можно прочитать в [3].

Упражнения

1. Модифицировать алгоритм поиска в глубину так, чтобы для каждого ребра печатался его тип – древесное или обратное.

2. Ориентированный граф $G = (V, E)$ называется полусвязным, если для всех пар вершин $u, v \in V$, либо существует путь из u в v, либо путь из v в u, либо и то и другое одновременно. Разработайте эффективный алгоритм для определения, является ли данный граф G полусвязным. Докажите корректность разработанного алгоритма и проанализируйте время его работы.

3. N шестеренок пронумерованы от 1 до N ($N \le 10$). Заданы M ($0 \le M \le 45$) соединений пар шестеренок в виде (i, j), $1 \le i < j \le N$ (шестерня с номером i находится в зацеплении с шестерней j). Можно ли повернуть шестерню с номером 1?

Если да, то найти количество шестерен, пришедших в движение.

Если нет, то требуется убрать минимальное число шестерен так, чтобы в оставшейся системе при вращении шестерни 1 во вращение пришло бы максимальное число шестерен. Указать номера убранных шестерен (если такой набор не один, то любой из них) и количество шестерен, пришедших в движение.

4. Имеется N человек и прямоугольная таблица A[1:N,1:N]; элемент $A[i, j]$ равен 1, если человек i знаком с человеком j, A[i, j] =A[j,i]. Можно ли разбить людей на 2 группы, чтобы в каждой группе были только незнакомые люди?

5. Сильно связанная компонента ориентированного графа $G(V, E)$ – это максимальное множество вершин $V' \subseteq V$, такое что для каждой пары вершин $u, v \in V'$ существуют пути из u в v и из v в u. Разработайте на основе поиска в глубину алгоритм выделения всех сильно связанных компонент ориентированного графа.

Как может измениться количество сильно связных компонентов графа при добавлении в граф нового ребра?

3. Кратчайшие пути

Задача о поиске кратчайшего пути довольно часто встречается на практике. Например, водителю нужно проехать из Ярославля в Мурманск и у него есть дорожный атлас с указанием расстояний между каждой парой пересечений дорог. Как найти кратчайший путь? Эта задача легко решается с помощью аппарата теории графов.

Итак, пусть дан граф $G = (V, E)$, ребрам (или дугам в случае ориентированного графа) которого приписаны веса (стоимости), задаваемые взвешенной матрицей смежности A. Задача о кратчайшем пути состоит в нахождении кратчайшего пути от заданной начальной вершины $s \in V(G)$ до заданной конечной вершины $t \in V(G)$, при условии, что такой путь существует, то есть вершины s и t принадлежат одной компоненте связности.

Веса (стоимости), приписанные ребрам, могут быть положительными, отрицательными или нулями. Единственное ограничение состоит в том, чтобы в графе не было циклов с суммарным отрицательным весом. Если такой цикл все же существует и u – некоторая его вершина, то, двигаясь от s к u, обходя затем этот цикл достаточно большое число раз и попадая наконец в t, получим путь со сколь угодно малым весом. Таким образом, в этом случае кратчайшего пути не существует.

Поэтому будем предполагать, что все циклы в G имеют неотрицательный суммарный вес. Отсюда также вытекает, что неориентированные дуги (ребра) графа G не могут иметь отрицательные веса.

3.1. Алгоритм Дейкстры

Алгоритм Дейкстры (1959 г.) находит кратчайшие пути от заданной начальной вершины *v_begin* до всех остальных вершин графа. Основная идея этого алгоритма: на каждом шаге пытаемся уменьшить кратчайшее расстояние до непросмотренных вершин, используя очередную вершину, длину пути к которой уменьшить уже нельзя. А именно: допустим, что кратчайший путь от вершины *v_begin* к некоторой вершине *u* графа *G* проходит через промежуточную вершину *y*. Очевидно, что этот путь должен содержать кратчайший путь от *v_begin* к *y,* так как в противном случае можно было бы уменьшить длину пути от *v_begin* к *u* за счет выбора более короткого пути от *v_begin* к *y.* Таким образом, сначала надо найти кратчайший путь к вершине *y*.

Для достижения этой цели определяется множество непросмотренных вершин. Первоначально в нем содержатся все вершины графа, кроме начальной. На каждом шаге из этого множества выбирается та из вершин, расстояние до которой от начальной меньше, чем для других оставшихся вершин. Текущие кратчайшие расстояния от *v_begin* до соответствующей вершины хранятся в массиве *Distance.* Далее пробуем с помощью ребер выбранной вершины *v_min* уменьшить длину пути до оставшихся непросмотренными вершин. Если это удается, то массив *Distance* корректируется. Алгоритм Дейкстры также использует массив *parent,* который содержит номера вершин – элемент *parent*[*k*] есть номер предпоследней вершины на текущем кратчайшем пути из начальной вершины в *k*-ю.

Код алгоритма представлен в листинге 7. Здесь предполагается, что в граф задан взвешенной матрицей смежности. Матрица расстояний *matr* является копией взвешенной матрицы смежности графа, только если дуги (ребра) не существует, то вместо нуля в соответствующую ячейку записывается большое число, равное «машинной бесконечности».

```
public  class  DijkstraAlgm
{ // Алгоритм Дейкстры поиска кратчайшего расстояния от
  // вершины v_begin до остальных вершин графа g
```

```
    int[] Distance;  // массив расстояний от начальной вершины
    int[] parent;    // массив предков на кратчайшем пути

    public  DijkstraAlgm(graph g, int v_begin)
    {
        Distance = new int[g.kol_vershn];
        parent = new int[g.kol_vershn];
        int[,] matr = new int[g.kol_vershn, g.kol_vershn];
// инициализация
for (int i = 0; i < g.matr_smeznosti.GetLength(0); i++)
            for (int j = 0; j < g.kol_vershn; j++)
            {
               matr[i, j] = g.matr_smeznosti[i, j];
               if (g.matr_smeznosti[i, j] == 0)
                      matr[i, j] = int.MaxValue;
            }
//множество непросмотренных вершин
          HashSet<int> mnvo_vn = new HashSet<int>();
          for (int i = 0; i < g.kol_vershn; i++)
          {
               mnvo_vn.Add(i);
// инициализация массива расстояний
               Distance[i] = matr[v_begin, i];
               if (Distance[i] < int.MaxValue)
                      parent[i] = v_begin;
           }
           Distance[v_begin] = 0;
           parent[v_begin] = -1;
           mnvo_vn.Remove(v_begin);
// основной цикл
```

```
        while (mnvo_vn.Count != 0)
        {
// ищем вершину из мн-ва с мин. расстоянием
            int min_d = int.MaxValue;  // мин. расстояние
            int v_min = -1;                    // номер выбранной вершины
            foreach (int u in mnvo_vn)
            {
                if (Distance[u] < min_d)
                {
                      min_d = Distance[u];
                      v_min = u;
                }
            }
// удаляем найденную вершину из мн-ва непросмотренных в-н
            if (v_min != -1)
                    mnvo_vn.Remove(v_min);
// пересчитываем расстояния
            foreach (int u in mnvo_vn)
            {
               if (matr[v_min, u] < int.MaxValue)
               {
                   Distance[u] = Math.Min(Distance[u],
                                 Distance[v_min] + matr[v_min, u]);
                   if (Distance[u] == (Distance[v_min] + matr[v_min, u]))
                   {
                          parent[u] = v_min;
                   }
               } }
      } // к основному циклу
Console.WriteLine(" Расстояния от вершины {0} до  ", v_begin);
      for (int i = 0; i < g.kol_vershn; i++)
```

```
    {
          Console.WriteLine("{0} = {1} ", i, Distance[i]);
     }
}
public   void  PrintPath(int v_begin, int v_end)
{ // метод печати последовательности вершин,
// составляющих кратчайший путь
        int u = parent[v_end];
        if (u == v_begin)
        {
           Console.Write(" {0}  {1} ", u, v_end);
           return;
        }
        PrintPath(v_begin, u);
         Console.Write("{0}  ", v_end);
  }
}
```

Листинг 7. Алгоритм Дейкстры

Трудоемкость алгоритма Дейкстры равна $O(n^2)$. Рассмотрим работу алгоритма на примере взвешенного ориентированного графа, изображенного на рис. 14.

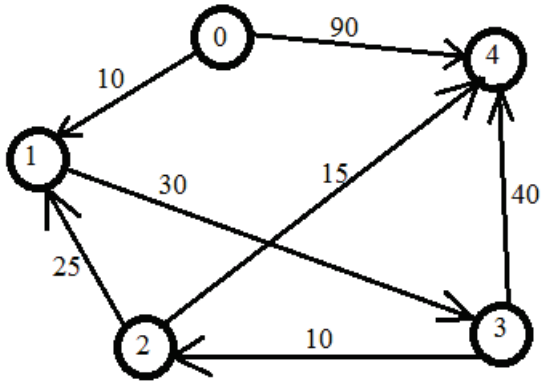


Рис. 14. Взвешенный орграф

Номер шага	Мн-во непросм. вершин	Тек. вершина	Distance				Parent				
			1	2	3	4	0	1	2	3	4
0	{1, 2, 3, 4}	-	10	∞	∞	90	-1	0	-1	-1	0
1	{2, 3, 4}	1	10	∞	40	90	-1	0	-1	1	0
2	{2, 4}	3	10	50	40	80	-1	0	3	1	3
3	{4 }	2	10	50	40	65	-1	0	3	1	2

После выполнения алгоритма кратчайший путь к каждой вершине можно найти с помощью обратного прохождения по предшествующим вершинам массива *Parent*.

Приведем неформальное доказательство корректности алгоритма: на каждом шаге выбирается еще не просмотренная вершина *v_min*, расстояние до которой от начальной вершины наименьшее из всех необработанных вершин. Затем при помощи дуг (ребер) вершины *v_min* уменьшается, если это возможно, расстояние до непросмотренных вершин. Очевидно, длину пути до тех вершин, которые уже просмотрены, уменьшить нельзя (действительно, для всех *u*, где *u* – уже просмотренная вершина, имеем *Distance*[*u*] < *Distance*[*v_min*] + *matr*[*u, v_min*], так как *Distance*[*u*] < *Distance*[*v_min*] – по правилу выбора очередной вершины в алгоритме, а все дуги имеют неотрицательный вес).

Далее, предположим для выбранной вершины длина пути не минимальна. Тогда существует путь с суммарным весом меньше, чем *Distance*[*v_min*]. Обозначим за *y* предпоследнюю вершину в этом пути. Получили, что *Distance*[*v_min*] > *Distance*[*y*] + *matr* [*y*, *v_min*], что противоречит шагу алгоритма.

Заметим, что если требуется построить путь от начальной вершины только до одной конечной, то при исключении конечной вершины из множества еще не просмотренных вершин можно завершать работу алгоритма.

3.2. Кратчайшие пути между всеми парами вершин

Общая задача нахождения кратчайших путей заключается в нахождении для каждой упорядоченной пары вершин (*v*, *u*) такого пути от вершины *v* в вершину *u*, что его длина будет минимальна среди всех возможных путей от *v* к *u*. Эта задача

может возникнуть, например, если имеется компьютерная сеть и известно время прохождения сетевого пакета между соседними компьютерами. Требуется узнать время прохождения пакета от каждого компьютера к каждому или максимальный временной интервал для прохождения пакета. Или, допустим, дан взвешенный орграф, который содержит время полета по маршрутам, связывающим определенные города, и необходимо построить таблицу, где приводилось бы минимальное время перелета из одного города в любой другой.

Подобные задачи можно решить, последовательно применяя алгоритм Дейкстры для каждой вершины, объявляемой в качестве начальной. Но существует прямой способ решения, использующий алгоритм Флойда.

Алгоритм Флойда использует матрицу *Distance* размера $n \times n$, где n – количество вершин в графе, в которой вычисляются длины кратчайших путей. Вначале каждый элемент матрицы *Distance* равен соответствующему элементу матрицы смежности, диагональные элементы равны 0. Если в графе дуга между вершинами *u* и *v* отсутствует, то *Distance*[*u, v*] =∞.

Над матрицей *Distance* выполняется *n* итераций. После *k*-й итерации *Distance*[*i, j*] содержит значение наименьшей длины путей из вершины *i* в вершину *j*, которые не проходят через вершины с номером, большим *k*. Другими словами, между концевыми вершинами пути *i* и *j* могут находиться только вершины, номера которых меньше или равны *k*.

На *k*-й итерации для вычисления матрицы *Distance* применяется следующая формула:

$$Distance\,[i, j] = \min\,(Distance\,[i, j], Distance\,[i, k] + Distance\,[k, j]).$$

Одновременно с матрицей расстояний ведется построение матрицы предков parent, которая необходима для востановления последовательности вершин, составляющих кратчайший путь. В элемент *parent*[*i, j*] записывается номер последней промежуточной вершины на пути от *i* к *j*.

Код алгоритма приведен в листинге 8.

```
public class FloydAlgm
{ // Алгоритм Флойда поиска кратчайших расстояний
//    между всеми парами вершин
      int[,] Distance;  // матрица кратчайших путей
      int[,] parent;    // матрица предков
      public   FloydAlgm(graph g)
      {
          int i, j, k; // индексы
// инициализация
          Distance = new int[g.kol_vershn, g.kol_vershn];
          parent = new int[g.kol_vershn, g.kol_vershn];
          for ( i = 0; i < g.kol_vershn; i++)
          {
             for (j = 0; j < g.kol_vershn; j++)
             {
               if (g.matr_smeznosti[i, j] == 0)  // если дуги нет, то
               {
                     Distance[i, j] = 10000; // длина пути = бесконечности
                     parent[i, j] = -1;
               }
               else  {
                     Distance[i, j] = g.matr_smeznosti[i, j];
                     parent[i, j] = i;
               }
             }
             Distance[i, i] = 0;
             parent[i, i] = i;
          }
// основной цикл
          for ( k = 0; k < g.kol_vershn; k++)
          {
```

```
            for (i = 0; i < g.kol_vershn; i++)
            {
               if (i == k) continue;
               for (j = 0; j < g.kol_vershn; j++)
               {
                  if ((j == i) || (k == j)) continue;
                  if (Distance[i, k] + Distance[k, j] < Distance[i, j])
                  {
                     Distance[i, j] = Distance[i, k] + Distance[k, j];
                     parent[i, j] = k;
                  }
               }
            }
         }
      }
//метод печати кратчайшего пути
    public void PrintPath(int v_begin, int v_end)
    {
        if (v_begin == v_end)
        {
          // Console.Write(" {0} ", v_begin);
        }
        else if (parent[v_begin, v_end] == -1)
        {
            Console.WriteLine(" не существует пути между”);
            Console.Write( “ {0} и {1} ", v_begin, v_end);
            return;
        }
        else PrintPath(v_begin, parent[v_begin, v_end]);
        Console.Write(" {0} ", v_end);
      }
```

```
// метод, возвращающий расстояние
      public int DistanceValue(int s, int t)
      {
         return Distance[s, t];
      }
}
```

Листинг 8. Реализация алгоритма Флойда

Время исполнения алгоритма Флойда равно $O(n^3)$. Это ничуть не лучше, чем n вызовов алгоритма Дейкстры, но на практике он работает эффективнее, так как циклы этого алгоритма очень короткие. Алгоритм Флойда примечателен еще и тем, что быстрее работает при представлении графа с помощью матрицы смежности.

Во многих задачах интерес представляет только сам факт существования пути, длиной не меньше единицы, от вершины i до вершины j. В качестве примера можно привести граф шантажиста, в котором наличие дуги от i к j означает, что лицо i владеет компроматом на персону j и может принудить ее сделать что угодно. Для лоббирования выгоднее нанять то лицо, которое имеет компромат на наибольшее количество людей. Этому человеку будет соответствовать та вершина в графе, из которой достижимо наибольшее количество других вершин. На языке теории графов такая задача называется построением транзитивного замыкания.

Транзитивным замыканием графа G называется ориентированный граф, такой что дуга между вершинами i и j существует тогда и только тогда, когда в графе существует путь от вершины i к j. Алгоритм Флойда можно приспособить для нахождения транзитивного замыкания. Но полученный в результате алгоритм еще до Флойда разработал С. Варшалл, поэтому в литературе часто алгоритм Флойда называют алгоритмом Флойда – Варшалла.

Если вес каждой дуги положить равным единице и выполнить алгоритм Флойда, то по матрице расстояний легко строится транзитивное замыкание: если $Distance[i, j] < \infty$, то дуга от вершины i к вершине j существует. Также легко выяснить, из какой вершины достижимо наибольшее количество вершин, для задачи о шантажисте.

Упражнения

1. Разработайте эффективный алгоритм для поиска самого дешевого пути в графе со взвешенными вершинами (стоимостью пути от вершины s к вершине t является сумма весов всех вершин, входящих в путь).

2. Диаметром графа называется максимальное кратчайшее расстояние между всеми парами вершин графа. Разработайте алгоритм, который определяет диаметр ориентированного графа.

3. В ориентированном взвешенном графе все вершины имеют положительный вес. Контуром называется ориентированный цикл. Разработайте алгоритм для поиска в орграфе контура с минимальным весом.

4. Разработайте алгоритм для поиска самого длинного пути из заданной вершины.

5. Дан взвешенный граф, у которого и вершины, и ребра имеют положительный вес. Если ребро отсутствует, то его вес равен бесконечности. Приведите алгоритм, который строит путь минимальной стоимости. Стоимость пути определяется как сумма весов ребер и вершин, через которые проходит путь.

6. *Корнем* ациклического орграфа называется вершина r такая, что существуют пути, исходящие из этой вершины и достигающие всех остальных вершин орграфа. Разработайте алгоритм, который определяет имеет ли данный ациклический орграф корень.

7. Пусть с каждым ребром орграфа связано число $r(u,v)$ ($0 \le r(u,v) \le 1$). Будем интерпретировать $r(u,v)$ как надежность – вероятность того, что сигнал успешно пройдет по каналу из u в v. Считая, что события прохождения сигнала по различным каналам независимы, предложите алгоритм для нахождения наиболее надежного пути между двумя данными вершинами.

8. Разработайте алгоритм для решения следующей задачи: на олимпиаду прибыло N человек. Некоторые из них знакомы между собой. Можно ли опосредованно перезнакомить их всех между собой? (Незнакомые люди могут познакомиться только через общего знакомого.)

4. Остов минимального веса

Рассмотрим следующую задачу: во взвешенном связном графе требуется найти остов минимального веса. Эта задача часто возникает при проектировании, например, линий электропередачи, трубопроводов, дорог и т. п., когда требуется заданные центры соединить некоторой системой каналов связи так, чтобы любые два центра были связаны либо непосредственно соединяющим их каналом, либо через другие центры и каналы и чтобы общая длина (или стоимость) каналов связи была минимальной. В этой ситуации заданные центры можно считать вершинами полного графа с весами ребер, равными длинам (стоимости) соединяющих эти центры каналов. Тогда искомая сеть будет кратчайшим остовным подграфом полного графа.

Очевидно, что этот кратчайший остовный подграф должен быть деревом.

Так как полный граф K_n содержит n^{n-2} различных остовных деревьев, то решение этой задачи перебором вариантов потребовало бы чрезвычайно больших вычислений даже при малых n. Существуют эффективные алгоритмы, решающие эту задачу, например алгоритм Дж. Краскала (1956 г.) и Р. Прима (1957 г.), применимые к произвольному связному графу.

4.1. Алгоритм Краскала

Общая формулировка задачи об остове минимального веса (о кратчайшем остове): в связном взвешенном графе G порядка n найти остов минимального веса. Алгоритм Краскала, решающий эту задачу, заключается в следующем.

1. Строим граф $T_1 = O_n + e_1$, присоединяя к пустому графу на множестве вершин $V(G)$ ребро $e_1 \in E(G)$ минимального веса.

2. Если граф T_i уже построен и $i < n$-1 , то строим граф $T_{i+1} = T_i + e_{i+1}$, где e_{i+1} – ребро графа G, имеющее минимальный вес среди ребер, не входящих в T_i и не составляющих циклов с ребрами из T_i.

Докажем, что алгоритм Краскала всегда приводит к остову минимального веса, а именно, что при $i < n$ -1 граф T_{i+1} можно построить и граф T_{n-1} является остовом минимального веса графа G.

Действительно, граф T_i имеет ровно i ребер и поэтому при $i < n$ -1 является несвязным. А так как граф G связен, то в нем есть, по меньшей мере, одно ребро, не составляющее циклов с ребрами графа T_i. Таким образом, нужное ребро e_{i+1} существует и граф T_{i+1} можно построить.

Теперь рассмотрим граф T_{n-1}. Поскольку T_{n-1} является графом порядка n и у него n-1 ребро, то это дерево. Остается доказать, что вес дерева T_{n-1} минимален. Предположим, что это не так и существует какой-то граф G, на котором алгоритм выдает неправильный ответ. Среди всех остовов графа G минимального веса выберем такой остов T_{min}, который имеет с деревом T_{n-1} максимальное количество общих ребер. Пусть $e_i = \{a,b\}$ – ребро дерева T_{n-1}, не содержащееся в T_{min} и имеющее минимальный номер среди ребер дерева T_{n-1}, не входящих в T_{min} (напомним, что в процессе построения дерева T_{n-1} его ребра получили номера 1, 2, . . ., n-1). При добавлении в дерево T_{min} ребра e_i получим цикл. В этом цикле есть ребро e, не входящее в дерево T_{n-1}. Заменив в дереве T_{min} ребро e на e_i, получим новый остов $T_{new} = T_{min} – e + e_i$. Но T_{min} – остов минимального веса, следовательно, вес дерева $w(T_{new}) \geq w(T_{min})$, отсюда вес ребра $w(e_i) \geq w(e)$.

С другой стороны, присоединяя ребро e к T_{i-1} (при $i = 1$ полагаем $T_{i-1} = O_n$), цикла не будет, поскольку ребра $e_1, e_2, ..., e_{i-1}, e$ входят в дерево T_{min}, и потому, если бы вес $w(e_i)$ был больше, чем $w(e)$, при построении дерева T_i было бы выбрано ребро e вместо e_i. Поэтому $w(e_i) = w(e)$ и $w(T_{new}) = w(T_{min})$.

Итак, T_{new} – остов минимального веса. Число ребер, общих для деревьев T_{new} и T_{n-1}, больше, чем число ребер, общих для T_{min} и T_{n-1}, что противоречит выбору дерева T_{min}. Полученное противоречие доказывает корректность алгоритма.

Реализация алгоритма Краскала приведена в листинге 9. Исходный граф задается списком смежности. Для проверки наличия цикла при добавлении ребра используется массив пометок *Mark*. Изначально каждая вершина помечена своим номером. Ребро-кандидат можно включать в дерево, если его вершины имеют разные пометки, то есть принадлежат разным компонентам связности. При добавлении ребра в остов инцидентные ему вер-

шины помечаются меньшей из двух пометок. Также корректируются пометки вершин, находящихся в той же компоненте связности.

```
internal  class rebro_grapha: IComparable
 {
     internal  int v { get; set; }  // номера вершин,
     internal  int u { get; set; }  // образующих ребро
     internal  int weight { get; set; } // вес ребра
// реализация интерфейса для сравнения ребер по весу
      public   int  CompareTo(object obj)
     {
        rebro_grapha rg = (rebro_grapha)obj;
        if (this.weight > rg.weight)            return 1;
        else if (this.weight == rg.weight)    return 0;
        else return -1;
     }
   }
public  class  AlgmKraskala
{ // алгоритм Краскала построения минимального остова
      List<rebro_grapha> sp_reber;  //список ребер графа
      int[] Mark;                          // массив пометок
      Graph OstTree;                     // остов графа
// конструктор, инициализирующий структуры данных
    public AlgmKraskala(Graph g)
    {
// преобразование графа в массив ребер
         sp_reber = new List<rebro_grapha>();
         Mark = new int[g.kol_v_n];
         for (int i = 0; i < g.kol_v_n; i++)
         {
            List<rebro> rlist = g.sp_reber[i];
```

```
            foreach (rebro r in rlist)
            {
              rebro_grapha rg = new rebro_grapha { v = i, u = r.v_na,
                                                   weight = r.ves };
               sp_reber.Add(rg);
            }
            Mark[i] = i;
          }
         sp_reber.Sort(); // сортировка списка ребер
        OstTree = new Graph(g.kol_v_n); // создание пустого графа
    }
    void AddRebro( rebro_grapha rg)
   { // метод добавления ребра в остов
         rebro e = new rebro{ v_na = rg.u, ves = rg.weight};
         OstTree.sp_reber[rg.v].Add(e);
    }
   internal Graph OstovTree()
   {// метод построения остовного дерева
         rebro_grapha first = sp_reber[0]; // самое легкое ребро
         sp_reber.RemoveAt(0);
         AddRebro(first); // включение первого ребра в остов
         Mark[first.v] = Math.Min(first.v, first.u);
         Mark[first.u] = Mark[first.v];
// основной цикл
         int kol_vo_reber_in_otree = 1; // кол-во ребер в остове
         int old_mark;
         while (kol_vo_reber_in_otree < (OstTree.kol_v_n-1))
         {
            rebro_grapha tek_r = sp_reber[0]; //очередное ребро
            sp_reber.RemoveAt(0);
// если включение ребра не приводит к циклу
```

```
            if (Mark[tek_r.v] != Mark[tek_r.u])
            { // включаем его в дерево
               AddRebro(tek_r);
               kol_vo_reber_in_otree++;
// изменяем массив пометок
               if (Mark[tek_r.v] > Mark[tek_r.u])
               {
                  old_mark = Mark[tek_r.v];
                  Mark[tek_r.v] = Mark[tek_r.u];
               }
               else
               {
                  old_mark = Mark[tek_r.u];
                  Mark[tek_r.u] = Mark[tek_r.v];
               }
               for (int i = 0; i < Mark.Length; i++)
                  if (Mark[i] == old_mark)
                     Mark[i] = Mark[tek_r.v];
            }
         }
         return OstTree;
      } }
```

Листинг 9. Реализация алгоритма Краскала

Трудоемкость алгоритма Краскала $O(nm)$. Действительно, упорядочивание m ребер занимает время $O(m\log m)$, основной цикл выполняется не более m раз, внутри него в цикле обрабатывается массив пометок *Mark* длины n, что и дает оценку трудоемкости.

4. 2. Алгоритм Прима

Алгоритм Прима отличается от алгоритма Краскала только тем, что на каждом шаге строится не просто ациклический граф, а дерево. Именно:

1. Выбираем ребро $e_1 = \{a, b\}$ минимального веса и строим дерево T_1, полагая $V(T_1) = \{a, b\}$, $E(T_1) = \{e_1\}$.

2. Если дерево T_i порядка i +1 уже построено и $i < n$-1, то среди ребер, соединяющих вершины этого дерева с вершинами графа G, не входящими в T_i, выбираем ребро e_{i+1} минимального веса. Строим дерево T_{i+1}, присоединяя к T_i ребро e_{i+1} вместе с его не входящим в T_i концом.

Доказательство корректности этого алгоритма аналогично приведенному выше. Код алгоритма приведен в листинге 10. Исходный граф задается списками смежности. Для хранения вершин остовного дерева используется множество *vershiniDereva*, так как для этой структуры данных проверка на вхождение в множество элемента равна константе и не зависит от размера множества. Можно было бы также использовать для этой цели массив пометок.

```
public  class  AlgmPrima
 {
     HashSet<int> vershiniDereva;  // множество вершин остова
     List<rebro_grapha> sp_reber; //список ребер исходного графа
     Graph OstTree;                    // остовное дерево
// метод добавления ребра в остовное дерево
     void AddRebro(rebro_grapha rg)
     {
        rebro e = new rebro { v_na = rg.u, ves = rg.weight };
        OstTree.sp_reber[rg.v].Add(e);
     }
// конструктор, инициализирующий структуры данных
     public AlgmPrima(Graph g)
     {
```

```
        vershiniDereva = new HashSet<int>();
        sp_reber = new List<rebro_grapha>();
        for (int i = 0; i < g.kol_v_n; i++)
        {  //создание списка  всех ребер графа
           List<rebro> rlist = g.sp_reber[i];
           foreach (rebro r in rlist)
           {
               rebro_grapha rg = new rebro_grapha { v = i, u = r.v_na,
                                                    weight = r.ves };
               sp_reber.Add(rg);
           }
        }
        sp_reber.Sort(); // упорядочивание списка ребер
        OstTree = new Graph(g.kol_v_n);
     }
// метод построения остовного дерева по алгоритму Прима
     internal Graph ostovTree()
     {
        int n = OstTree.kol_v_n;
// самое легкое ребро включается в остовное дерево
        rebro_grapha first = sp_reber[0];
        sp_reber.RemoveAt(0);
        AddRebro(first);
        vershiniDereva.Add(first.v);
         vershiniDereva.Add(first.u);
// основной цикл
        while (vershiniDereva.Count < n)
        {
           bool find = false; // флаг = выбрано ребро для дерева
           for (int i = 0; (i < sp_reber.Count) && (!find); i++)
           {
```

```
            rebro_grapha tek_rebro = sp_reber[i];
            if ((vershiniDereva.Contains(tek_rebro.v) &&
               !vershiniDereva.Contains(tek_rebro.u)) ||
            (!vershiniDereva.Contains(tek_rebro.v) &&
               vershiniDereva.Contains(tek_rebro.u)))
            {
                find = true;
                AddRebro(tek_rebro);
                sp_reber.RemoveAt(i);
                vershiniDereva.Add(tek_rebro.v);
                vershiniDereva.Add(tek_rebro.u);
            }
          }
        }
        return OstTree;
    }
}
```

Листинг 10. Реализация алгоритма Прима

Трудоемкость алгоритма Прима $O(nm)$. Действительно, основной цикл выполняется n раз, а внутри этого цикла просматривается список из m ребер.

На рис. 15 посередине приведен исходный граф, а справа и слева – его минимальные остовные деревья, построенные по алгоритмам Краскала и Прима. Числа на ребрах остовных деревьев обозначают последовательность их добавления. Легко видно, что остовные деревья абсолютно одинаковы, но порядок добавления в них ребер различен.

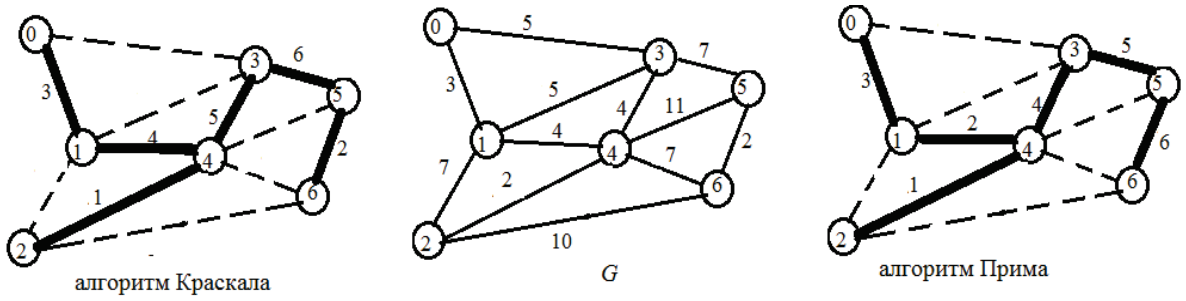


Рис. 15. Граф *G* и его минимальные остовные деревья

4.3. Разновидности остовных деревьев

Некоторые задачи приводят к необходимости построить остов не минимального, а максимального веса, например к нахождению максимального остовного дерева сводится проблема активации задач.

Рассмотрим ситуацию, когда на входе ЭВМ имеется некоторое множество задач, имеющих общие страницы в памяти. Требуется определить такую последовательность пропуска этих задач, чтобы минимизировать пересылки страниц в памяти. Для решения построим граф, вершинами которого будут служить задачи, а две вершины будут соединены ребром тогда и только тогда, когда они соответствуют задачам, имеющим общие страницы памяти. Каждому ребру сопоставляется вес, зависящий от числа общих страниц. Искомая последовательность выполнения задач будет задаваться остовным деревом максимального веса. К этой задаче также применимы алгоритмы Краскала и Прима, если изменить знак веса каждого ребра на противоположный.

Заметим, что большинство алгоритмов на графах нельзя так легко адаптировать на работу с отрицательными числами.

Если требуется найти остовное дерево с минимальным произведением весов ребер, то, учитывая, что log(ab) = log(a) + log(b), минимальное остовное дерево графа, в котором веса ребер заменены их логарифмами, дает нужное нам решение. Правда, вес ребер обязательно должен быть положительным.

С задачей об остове минимального веса тесно связана *задача Штейнера*. На практике эта задача возникает при проектировании коммуникационных сетей, когда необходимо соединить данное множество станций, используя кабель наименьшей протяженности. Естественно сопоставить станции вершинам графа, а

кабели между станциями – ребрам. Задача Штейнера отличается от задачи построения остова минимального веса в графе тем, что в этот граф разрешается вносить новые вершины – точки Штейнера. Их можно добавлять столько, сколько потребуется, чтобы дерево, их соединяющее, имело минимальный вес.

На рис. 16 а представлен исходный граф, представляющий собой равносторонний треугольник с длиной стороны, равной единице; рядом (рис. 16 б) – его минимальное остовное дерево. Его длина равна 2. И на рис. 16 в – дерево Штейнера исходного графа, которое соединяет все вершины графа при помощи еще одной, дополнительной. Его длина равна $\sqrt{3}$.

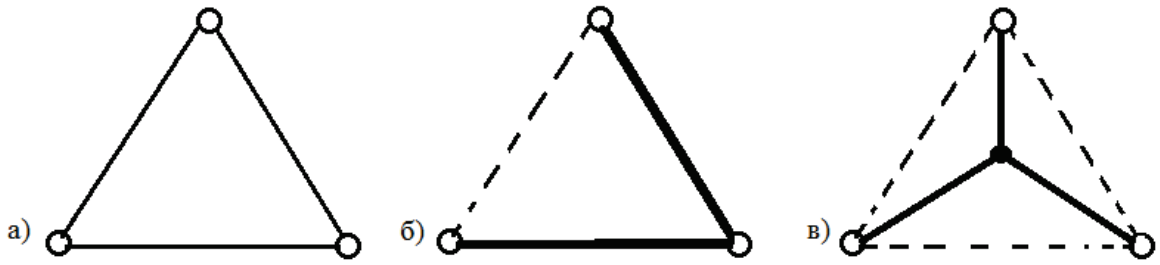


Рис. 16. Граф, его остовное дерево и дерево Штейнера

Какие-либо эффективные алгоритмы, решающие задачу Штейнера, неизвестны.

Упражнения

1. Объясните, могут ли алгоритмы Краскала и Прима выдавать разные минимальные остовные деревья.

2. Дан взвешенный граф G и его минимальное остовное дерево T. В G добавили ребро E с весом W. Разработайте алгоритм, который за минимальное число операций строит остовное дерево графа $G + E$.

3. Дан взвешенный граф G и его минимальное остовное дерево T. В G вес всех ребер увеличили на d. Объясните, останется ли T минимальным остовом графа G.

4. Укажите эффективный алгоритм для такой задачи: для данного взвешенного графа $G = (V, E, W)$ найти наилучшее покры-

вающее дерево, критерий «качества» дерева – вес его самого тяжелого ребра (который должен быть как можно меньше).

5. Пусть минимальный остов уже построен. Как быстро можно найти новый минимальный остов, если добавить к графу новую вершину и инцидентные ей ребра? Напишите алгоритм, решающий эту задачу.

6. Укажите эффективный алгоритм поиска второго по величине покрывающего дерева.

7. Разработайте алгоритм решения следующей задачи. Имеется N городов. Для каждой пары городов (i, j) можно построить дорогу, соединяющую эти два города и не заходящие в другие города. Стоимость такой дороги $A(i, j)$. Вне городов дороги не пересекаются. Требуется найти самую дешевую систему дорог, позволяющую попасть из любого города в любой другой. Результаты задавать таблицей B, где $B[i, j] = 1$ тогда и только тогда, когда дорогу, соединяющую города i и j, следует строить.

5. Циклы в графах

В этой лекции рассматриваются вопросы, связанные с существованием в графе самых различных циклов. Задачи, связанные с циклическим обходом графа очень часто встречаются на практике, например, при проверке оборудования, профилактических работах в коммуникационных сетях и т. д.

5.1. Эйлеров цикл

Цикл в графе называется *эйлеровым*, если он содержит все ребра графа. Связный граф, в котором есть эйлеров цикл, называется эйлеровым графом.

Такой граф можно нарисовать, не отрывая карандаша от бумаги и не повторяя линий.

Определить, содержит ли граф эйлеров цикл, достаточно просто. Полную характеристику эйлерова графа дает теорема, доказанная Л. Эйлером еще в 1736 г.

Теорема. *Связный неориентированный граф является эйлеровым тогда и только тогда, когда степени всех его вершин четны.*

Естественно возникает вопрос: как найти хотя бы один эйлеров цикл в эйлеровом графе G, т. е. как занумеровать ребра графа

числами 1, 2, ..., $m(G)$ с тем, чтобы номер, присвоенный ребру, указывал, каким по счету это ребро проходится в эйлеровом цикле? Ответ на этот вопрос дает алгоритм Флери. Ребра нужно нумеровать, придерживаясь следующих двух правил:

1. Начиная с произвольной вершины u присваиваем произвольному ребру $\{u, v\}$ номер 1. Затем вычеркиваем ребро $\{u, v\}$ и переходим в вершину v.

2. Пусть w – вершина, в которую мы пришли в результате выполнения предыдущего шага, и k – номер, присвоенный некоторому ребру на этом шаге. Выбираем любое ребро, инцидентное вершине w, причем мост выбираем только в том случае, когда нет других возможностей; присваиваем выбранному ребру номер $k + 1$ и вычеркиваем его.

Этот процесс заканчивается, когда все ребра графа вычеркнуты, т. е. занумерованы.

Дадим теперь обоснование алгоритма. Прежде всего заметим, что поскольку степень каждой вершины графа G четна, то алгоритм может закончить работу только в той вершине, с которой начал. Поэтому он строит некоторый цикл C, и надо только доказать, что этот цикл включает все ребра графа G.

Предположим, что это не так, и пусть G' – связная компонента графа $G \setminus E(C)$, не являющаяся изолированной вершиной. Рассмотрим множество A ребер цикла C, инцидентных вершинам, вошедшим в G'. Ясно, что $A \neq \emptyset$. Пусть a – ребро из A, получившее в процессе работы алгоритма наибольший номер, т. е. вычеркнутое последним среди ребер A. Тогда, как легко видеть, ребро a к моменту вычеркивания было мостом в графе. Однако это противоречит правилу выбора очередного ребра.

Реализация алгоритма Флери основана на поиске в глубину. Пусть граф G, удовлетворяющий условию теоремы, задается матрицей смежности. Будем обходить граф методом поиска в глубину, при этом каждое просмотренное ребро удаляется. При обнаружении вершины, все ребра которой удалены, ее номер записывается в стек и просмотр продолжается от предыдущей вершины. Обнаружение вершин с нулевым числом ребер говорит о том, что найден цикл. Его можно удалить, четность степеней вершин при этом не изменится. Процесс продолжается до тех

пор, пока есть ребра. В стеке после этого будут записаны номера вершин графа в порядке, соответствующем эйлерову циклу.

Код алгоритма приведен в листинге 11.

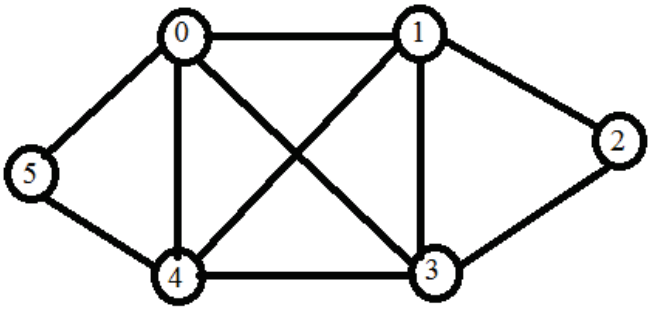
```
public class EulerCycleSearch
{ // алгоритм Флёри поиска Эйлерова цикла в графе
    Stack<int> cycleVertex; // стек для хранения вершин цикла

    public EulerCycleSearch(graph g)
    {
        cycleVertex = new Stack<int>();
        DFS_traversal(g, 0); // начало обхода графа
        Console.WriteLine("Вершины эйлерова цикла");
        while( cycleVertex.Count>0)
        {
             Console.Write("{0}  ",cycleVertex.Pop());
        }
        Console.WriteLine();
    }
    void DFS_traversal(graph g, int v)
    {// рекурсивный метод обхода
        for( int i = 0; i < g.kol_vershn; i++)
        {// если вершины смежны  и в i еще не были
            if (g.matr_smeznosti[v, i] != 0)
            {  // удаление пройденного ребра
                g.matr_smeznosti[v,i]=0;
                g.matr_smeznosti[i,v]=0;
                DFS_traversal(g, i); //рекурсивный вызов
            }
         }
cycleVertex.Push(v);// сохранение вершины в стеке
```

```
}
}
```

Листинг 11. Реализация алгоритма Флери поиска эйлерова цикла в графе

Временная сложность алгоритма Флери совпадает с сложностью алгоритма поиска в глубину и равна $O(n^2)$. На рис. 17 изображен граф G и приведена последовательность вершин, составляющих эйлеров цикл, которую находит алгоритм.



Последовательность вершин, составляющих эйлеров цикл:
0 1 2 3 0 4 1 3 4 5 0.

Рис. 17. Граф G и его эйлеров цикл

Все сказанное в этом параграфе о графах может быть перенесено и на мультиграфы.

5.2. Задача китайского почтальона

На практике задача о поиске эйлерова цикла в графе равнозначна задаче о поиске самого короткого маршрута, который проходит по каждому ребру графа один раз. Таковы задачи об оптимальном маршруте для снегоуборочной машины, или мусоровоза, или почтальона. Но сеть дорог в городе редко удовлетворяет теореме Эйлера. Поэтому приходится решать более общую задачу о нахождении кратчайшего цикла, который проходит по каждому ребру графа хотя бы один раз. Эта задача впервые была поставлена китайским ученым Кваном, что и определило ее название – задача китайского почтальона.

Оптимальный маршрут китайского почтальона можно легко найти, если добавить дополнительные ребра в исходный граф G так, чтобы сделать его эйлеровым. Для этого нужно найти кратчайший

путь между каждой парой вершин с нечетной степенью. Добавив фиктивные ребра вдоль каждого такого пути, получим граф, у которого степени всех вершин будут четными. Задача поиска наилучшего множества кратчайших путей для добавления к графу G сводится к задаче о поиске совершенного паросочетания минимального веса в особом графе G'. Подробно этот алгоритм описан в книге [2].

5.3. Гамильтонов цикл и задача коммивояжера

Граф называется *гамильтоновым*, если в нем имеется простой цикл, содержащий каждую вершину этого графа. Сам этот цикл также называется *гамильтоновым*.

Слово «гамильтонов» в этих определениях связано с именем известного ирландского математика У. Гамильтона, которым в 1859 г. предложена следующая игра «Кругосветное путешествие». Каждой из двадцати вершин додекаэдра приписано название одного из крупных городов мира. Требуется, переходя от одного города к другому по ребрам додекаэдра, посетить каждый город в точности один раз и вернуться в исходный город.

Эта задача, очевидно, сводится к отысканию в графе додекаэдра (рис. 18) простого цикла, проходящего через каждую вершину этого графа.

Несмотря на внешнее сходство постановок, задачи распознавания эйлеровости и гамильтоновости графа принципиально различны. Легко узнать, является ли граф эйлеровым и, в случае положительного ответа, алгоритм Флёри позволяет достаточно быстро построить один из эйлеровых циклов. Ответить на вопрос, является ли данный граф гамильтоновым, как правило, очень трудно.

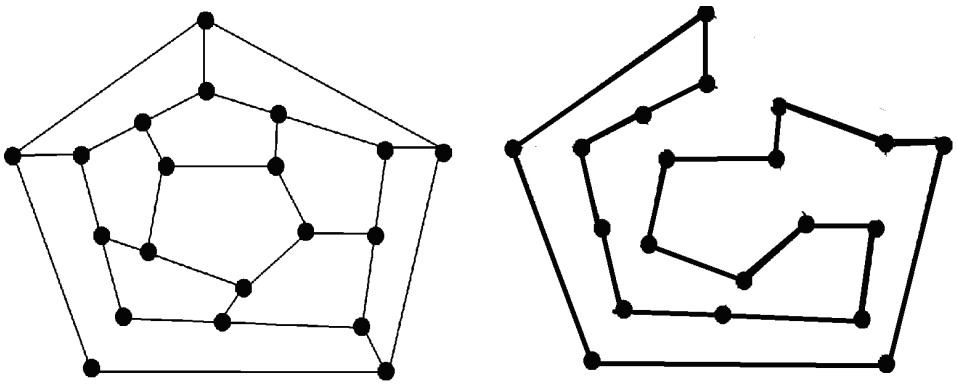


Рис. 18. Граф додекаэдра и его гамильтонов цикл

Практическое применение задача о поиске гамильтонова цикла в графе находит, например, в задачах распознавания образов. Там вершины графа соответствуют допустимым символам, а ребра соединяют те символы, которые могут быть соседними. Самый длинный путь в этом графе будет наилучшим кандидатом для правильной интерпритации. Поиск самого длинного пути или цикла тесно связан с задачей о гамильтонове цикле.

Хорошо известная задача коммивояжера состоит в следующем: коммивояжер должен посетить каждый из заданных городов по одному разу, выехав из некоторого из этих городов и вернувшись в него же. Требуется найти кратчайший маршрут, зная расстояния между каждой парой городов.

Математическая постановка этой задачи: в полном взвешенном графе требуется найти гамильтонов цикл минимального веса. Под весом цикла понимается сумма весов составляющих его ребер.

Конечно, для решения этой задачи существует простой алгоритм – полный перебор всех возможных вариантов. Однако такой подход, как правило, неприемлем из-за чрезвычайно большого числа этих вариантов (если число городов равно n, то число всех возможных обходов n^n). Поэтому вызывает интерес не просто алгоритм, а эффективный алгоритм.

Многие задачи, такие как задача коммивояжера, с вычислительной точки зрения представляются достаточно трудными. Для их решения до сих пор не найдено полиномиальных алгоритмов. Возможно, что таких алгоритмов не существует, но пока это не доказано.

5.4. Классы сложности P и NP

Будем рассматривать лишь задачи, сформулированные в так называемом распознавательном варианте, когда решение задачи заключается в получении ответа «да» или «нет». Таковы, например, задачи распознавания изоморфизма, гамильтоновости, эйлеровости графов. На практике чаще требуется решить оптимизационную задачу, т. е. выбрать из множества допустимых решений одно, вес или стоимость которого минимальна. Например, найти минимальное остовное дерево. Каждой оптимизационной задаче можно сопоставить семейство ее

распознавательных вариантов следующим образом: определить, существует ли решение с весом меньше некоторого c. Далее будем рассматривать только распознавательные задачи.

P – это класс распознавательных задач, каждая из которых может быть решена некоторым алгоритмом за полиномиальное время.

Понятие полиномиально разрешимой задачи принято считать уточнением идеи «практически разрешимой» задачи, так как, во-первых, полиномиальные алгоритмы действительно работают довольно быстро. Во-вторых, класс полиномиально разрешимых задач обладает естественными свойствами замкнутости: композиция двух полиномиальных алгоритмов (выход первого алгоритма подается на вход второго) также работает полиномиальное время.

Все рассмотренные выше задачи, независимо от того, установлена их принадлежность классу P или нет, обладают одним общим свойством: существует полиномиальный алгоритм, доказывающий, что имеет место ответ «да» по соответствующему входу задачи. Например, пусть задача состоит в выяснении, является ли граф гамильтоновым, и пусть поступающий на вход граф G гамильтонов, т. е. в графе G имеется гамильтонов цикл C. Тогда доказательство гамильтоновости графа G заключалось бы в проверке включения $C \subseteq G$, что можно сделать за $O(n)$ операций. Но если граф не является гамильтоновым, то нельзя утверждать, что существует полиномиальное доказательство этого факта, так как для перебора всех возможных простых циклов потребуется $O((n(G)\text{-}1)!)$ операций.

Назовем *проверяющим алгоритмом* алгоритм A с двумя аргументами: первый аргумент – входная строка, а второй – сертификат. Алгоритм A с двумя аргументами допускает вход x, если существует сертификат y, для которого $A(x, y) = 1$. Другими словами, алгоритм A проверяет задачу L, если для любого входа x задачи L найдется сертификат y, с помощью которого A сможет проверить, что в задаче L получен положительный ответ, а для входных данных, предполагающих отрицательный ответ, такого сертификата нет. Например, в задаче о гамильтоновом цикле сертификатом была последовательность вершин, образующая гамильтонов цикл.

NP – это класс всех распознавательных задач, для которых существуют проверяющие алгоритмы, работающие полиномиальное время, причем длина сертификата также ограничена некоторым полиномом.

Сокращение *NP* происходит от термина недетерминированно разрешимых за полиномиальное время, так как первоначально класс *NP* определялся с помощью недетерминированных вычислений.

Нетрудно видеть, что $P \subseteq NP$. Действительно, если есть полиномиальный алгоритм, решающий задачу, то легко построить проверяющий алгоритм для той же задачи – проверяющий алгоритм может просто игнорировать свой второй аргумент (сертификат).

Список проблем третьего тысячелетия открывается вопросом $P = NP$? Большинство исследователей считают, что $P \neq NP$. Интуитивно класс *P* можно представить себе как класс задач, которые можно быстро решить, а класс *NP* – как класс задач, которые можно быстро проверить. На практике решить самому задачу намного труднее, чем проверить готовое решение, особенно если время работы ограничено. Или можно считать, что в классе *NP* имеются задачи, которые нельзя решить за полиномиальное время.

Как оказалось из $P \neq NP$ следует, что ни одна из «трудных» задач не имеет полиномиального алгоритма, а из существования такого алгоритма для одной из них следует $P = NP$.

Изложение соответствующих результатов опирается на понятие сводимости одной задачи к другой, т. е. из решения одной задачи можно получить решение другой. Говоря неформально, задача *Q* сводится к задаче *Q*', если задачу *Q* можно решить для любого входа, считая известным решение задачи *Q*' для какого-то другого входа. Например, задача решения линейного уравнения сводится к задаче решения квадратного уравнения (линейное уравнение можно превратить в квадратное, добавив фиктивный старший член). Если задача *Q* сводится к задаче *Q*', то любой алгоритм, решающий *Q*', можно использовать для решения задачи *Q*.

Задача *Q сводится* к задаче *Q*', если существует полиномиальный алгоритм *F*, который, будучи примененным ко всякому входу

x задачи Q, строит некоторый вход $F(x)$ задачи Q', и вход x имеет ответ «да» тогда и только тогда, когда ответ «да» имеет вход $F(x)$.

Нетрудно показать, что если задача Q сводится к задаче Q' и $Q' \in P$, то и $Q \in P$. Действительно, пусть A – алгоритм решения задачи Q' и полиномы $P_1(n)$ и $P_2(n)$ таковы, что $O(P_1(n))$ и $O(P_2(n))$ – трудоемкости алгоритмов F и A соответственно. Рассмотрим теперь алгоритм B решения задачи Q, который состоит из двух этапов. На первом этапе вход x задачи Q преобразуется алгоритмом F во вход $F(x)$ задачи Q'. На втором этапе алгоритм A применяется ко входу $F(x)$. Согласно определению F, алгоритм A сообщит ответ «да» тогда и только тогда, когда ответ «да» имеет вход x, то есть алгоритм B действительно решает задачу Q. Выясним теперь его сложность. Если длина входа x равна n, то $F(x)$ будет построен за время $O(P_1(n))$ и его длина – $O(P_1(n))$. Алгоритм A, будучи примененным ко входу $F(x)$, затратит время $O(P_2(P_1(n)))$. Таким образом, сложность алгоритма B есть $O(P_1(n)) + O(P_2(P_1(n)))$. Поскольку суперпозиция и сумма полиномов также являются полиномом, то алгоритм B – полиномиальный.

Задачу Q назовем *NP-полной*, если $Q \in NP$ и любая задача из NP сводится к Q. Очевидно, что если хотя бы одна NP-полная задача входит в P, то $P = NP$. Таким образом, гипотеза $P \neq NP$ означает, что NP-полные задачи не могут быть решены за полиномиальное время. Возможно, когда-нибудь будет найден полиномиальный алгоритм решения NP-полной задачи и тем самым доказано, что $P = NP$. Пока это никому не удалось, и поэтому доказательство NP-полноты некоторой задачи служит убедительным аргументом в пользу того, что она является практически неразрешимой.

В настоящее время известен значительный список NP-полных задач. Это и задача о поиске числа независимости, плотности графа, гамильтоновости, изоморфного подграфа.

5.5. Решение NP-полных задач

Допустим, доказано, что некоторая задача является NP-полной, то есть алгоритма с полиномиальным временем исполнения для ее решения при всех входных данных не существует. Однако

такую задачу всё равно надо решать. Существует три варианта решения:

- алгоритмы, эффективные для средних случаев задачи. Например, поиска с возвратом, где выполняются значительные отсечения;
- эвристические алгоритмы. Примером может служить «жадный» алгоритм. Для задачи коммивояжера он формулируется так: иди в ближайшую непройденную вершину. Эвристические алгоритмы позволяют быстро найти решение, но без гарантии, что это решение будет наилучшим;
- аппроксимирующие алгоритмы, которые дают гарантию, что найденное решение будет отличаться от оптимального решения не более чем на определенную погрешность на любом входе задачи.

Упражнения

1. Сформулируйте критерий существования эйлерова цикла для ориентированных графов.
2. Реализуйте алгоритм, решающий задачу китайского почтальона.
3. Приведите пример, когда «жадный» алгоритм решения задачи коммивояжера дает далеко не оптимальное решение.
4. Разработайте алгоритм, который находит все циклы в графе. Оцените его трудоемкость.

6. Независимые множества и покрытия

6.1. Независимые множества

Независимым подмножеством графа G называется такое подмножество его вершин V_0, в котором никакие две вершины не соединены ребром.

Иными словами, если $S \subseteq V(G)$ и S независимо в графе G, то порожденный подграф $G(S)$ является пустым. Очевидно, что если при этом $S' \subseteq S$, то S' также независимое множество.

Независимое множество называется *максимальным*, если оно не является собственным подмножеством некоторого другого независимого множества.

Максимальное по мощности независимое множество называется *наибольшим*. Ясно, что наибольшее независимое множество является максимальным. Обратное, в общем случае, неверно.

К отысканию наибольшего независимого множества вершин в графе сводится, например, известная задача о восьми ферзях, которую связывают с именем К. Гаусса: требуется так расставить на шахматной доске наибольшее число ферзей, чтобы они не атаковали друг друга.

Таких ферзей, очевидно, может быть не более восьми, так как никакие два из них не должны находиться на одной вертикали или горизонтали. Рассмотрим граф, вершины которого соответствуют клеткам доски, а ребра – парам клеток, лежащих на одной вертикали, горизонтали или диагонали. Ясно, что требуемой в задаче расстановке ферзей соответствует наибольшее независимое множество в этом графе.

Число вершин в наибольшем независимом множестве графа G называется числом независимости или неплотностью этого графа и обозначается $\alpha_0(G)$.

Например, для пустого графа $\alpha_0(O_n) = n$, для полного $\alpha_0(K_n) = 1$. Для графа G, изображенного на рис. 20, $\alpha_0(G) = 4$, множества вершин {1, 2, 3, 7}, {1, 2, 3, 8}, {2, 3, 5, 7} и {2, 3, 5, 8} являются наибольшими независимыми, а {4, 7} – максимальное независимое множество, не являющееся наибольшим.

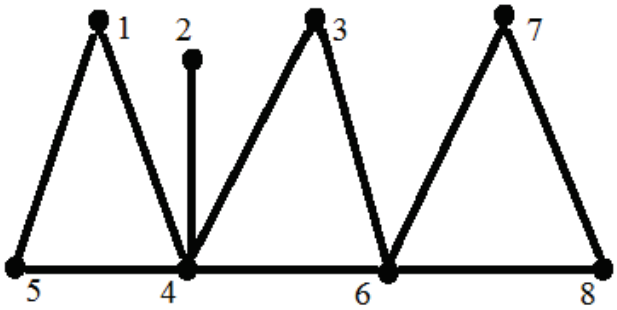


Рис. 20

Независимые множества вершин графа имеют самые разнообразные применения. Рассмотрим одно из применений независимости в теории информации. Возникающую здесь ситуацию

можно упрощенно описать следующим образом. Источник информации посылает сообщения, являющиеся последовательностями сигналов из множества $X = x_1, x_2, \ldots, x_m$. При передаче возникают (например, вследствие помех) искажения сигналов. Поэтому на принимающей станции некоторые сигналы могут быть поняты как другие, т. е. перепутаны. Рассмотрим граф G, у которого $V(G) = X$ и $\{x_i, x_j\} \in E(G)$, если и только если x_i и x_j могут быть перепутаны. Тогда, чтобы получить безошибочный код, т. е. исключить перепутывания, следует пользоваться сигналами из независимого подмножества вершин графа G. Стремление получить максимальное количество таких сигналов приводит к задаче отыскания наибольшего независимого множества вершин в графе G.

К поиску наибольшего независимого множества вершин сводится задача о выборе места для точек обслуживания. Допустим, требуется разместить магазины некоторой сети так, чтобы избежать конкуренции между ними. Тогда для решения этой задачи можно создать граф, вершины которого сответствуют возможным местам расположения магазинов, и вершины соединены ребром в том случае, когда магазины «мешают» друг другу. Наилучшее расположение магазинов будет соответствовать наибольшему независимому множеству этого графа.

Между элементами независимого множества отсутствуют конфликты, и поэтому их часто используют в задачах календарного планирования.

Задача о поиске наибольшего независимого множества тесно связана с двумя другими *NP*-полными задачами: с задачей о наибольшей клике (максимальное независимое множеств графе G является максимальной кликой в дополнительном графе к G, поэтому алгоритмически эти задачи идентичны) и с задачей вершинной раскраски. Задача вершинной раскраски заключается в разбиении множества вершин графа на подмножества таким образом, чтобы смежные вершины были разных цветов. Понятно, что вершины одного цвета составляют независимое множество.

Самый простой эвристический алгоритм поиска независимого множества можно описать так: выбираем в графе вершину минимальной степени, помещаем ее в независимое множество.

Из исходного графа удаляем эту вершину и смежные с ней. Продолжаем процесс с усеченным графом до тех пор, пока граф не окажется пустым. Ниже приведен полиномиальный алгоритм аппроксимации независимого множества, имеющий лучшую на сегодняшний день погрешность.

6.2. Алгоритм аппроксимации максимального независимого множества методом исключения подграфов

Авторами данного алгоритма являются Р. Боппана и М. Халлдорсон. Доказательство оценки погрешности можно посмотреть в статье [6].

Введем обозначения: $I(G)$ – максимальное независимое множество, $C(G)$ – максимальная клика.

Под $N(v)$, где $v \in G$, будем понимать подграф графа G, состоящий из всех вершин и ребер, инцидентных вершине v.

Аналогично, под $\overline{\mathrm{N}}(v)$, где $v \in G$, будем понимать подграф графа G, состоящий из всех вершин и ребер, не инцидентных вершине v.

Для нахождения максимального независимого множества используется рекурсивный метод, который состоит в следующем: выберем вершину v и занесем её в независимое множество. Построим подграф $\overline{N}(v)$ (из этого подграфа будут выбираться следующие вершины в независимое множество).

Аналогично сформулируем метод для поиска максимальной клики, с той разницей, что для вершины v ищем подграф $N(v)$, из которого в дальнейшем будут выбираться вершины в максимальную клику.

Заметим, что при неудачном выборе начальной вершины можно исключить из рассмотрения значительные независимые компоненты (клики). Поэтому предлагается следующая модификация алгоритма. Как и прежде, выбираем вершину v, так же находим $\overline{N}(v)$, но на этот раз еще строим граф $N(v)$. Величину независимого множества будем определять как максимум из величин чисел независимости подграфов $N(v)$ и $v \cup \overline{N}(v)$. Аналогичный алгоритм работает и для максимальных клик.

Для более точной аппроксимации независимого множества введем правило выбора вершины v. Вершину будем выбирать с

наименьшей степенью, т. к. в этом случае окрестность подобной вершины может содержать более значительное независимое множество. Для максимальной клики вводится аналогичное правило. Здесь в качестве *v* выбирается вершина с наибольшей степенью.

Фактически алгоритм строит древовидную структуру подзадач, превращая граф в двоичное дерево. Рассмотрим некоторый внутренний узел *t* этого дерева. Все левые потомки узла *t* – это вершины, смежные с *t*, а все правые потомки – несмежные с *t* вершины. Таким образом, независимое множество вершин, найденное алгоритмом, связано с маршрутом в дереве с самым большим количеством правых потомков. Следовательно, его размер является точным максимальным количеством правых потомков в любом маршруте в дереве плюс одно. Аналогично, размер найденной клики является максимальным количеством левых потомков в любом маршруте плюс одно. Введенное правило выбора вершины позволяет удлинить маршрут в дереве с правыми или левыми потомками в зависимости от того, какую вершину выбираем с минимальной или максимальной степенью вершины для более точной аппроксимации независимого множества или клики.

Например, рассмотрим входной граф *G*, не содержащий треугольники. Несомненно, алгоритм не может найти клики размера более чем 2, и, следовательно, все маршруты в дереве имеют не более одного левого потомка. Из этого следует также, что самый правый маршрут имеет $\sqrt{n}$ узлов либо в дереве менее $\sqrt{n}$ маршрутов. Но и тогда один из них имеет более чем $\sqrt{n}$ узлов. В любом случае алгоритм находит независимое множество размера не менее чем $\sqrt{n}$.

Выполнение алгоритма, который находит клику размера *s* и независимое множество размера *t,* соответствует двоичному дереву, где наибольшее число левых потомков в некотором маршруте – $(s - 1)$ и наибольшее количество правых потомков – $(t - 1)$. Пусть $r(s, t)$ обозначает минимальное целое *p* такое, что все деревья размера *p* имеют маршруты с большим количеством либо левых, либо правых потомков. Эта величина больше, чем размер самого большого дерева, в котором нет маршрута с $(s - 1)$ левым потомком или $(t - 1)$ правым. Но эта величина меньше, чем количество листьев в таком дереве. Поскольку каждый лист име-

ет связанный уникальный маршрут, то в дереве может быть не более чем $C^{t-1}_{(s-1)+(t-1)}$ таких узлов. Следовательно, $r(s, t) < C^{t-1}_{s+t-2}$.

Пусть $R(s,t)$ (число Рамсея) обозначает минимальное целое n так что все графы размера n или содержат независимое множество размера t или клику размера s. Верхнюю оценку числа Рамсея доказали Эрдеш и Шекереш [7] в 1935 г.:

$$R(s,t) \le r(s,t) \le \quad C^{t-1}_{s+t-2}.$$

Отсюда следует, что описанный алгоритм (авторы назвали его Ramsey) находит независимое множество I и клику C такие, что $|I| \cdot |C| \ge \frac{1}{4} (\log n)^2$. Реализация алгоритма Ramsey приведена в листинге 12.

```
internal class rebro // класс ребро
 {
      public int v_na { get; set; }
 }
 internal class vershina // класс, представляющий вершину в графе
 {
      public int degree { get; set; }    // степень вершины
     internal List<rebro> sp_reber;  // список инцидентных вершин
      public vershina()
      {
         sp_reber = new List<rebro>();
      }
}
public class Graph
 {// класс, представляющий граф, заданный списками смежности
      internal Dictionary<int, vershina> sp_vershin;
      internal int kol_v_n { get; set; }
      public Graph()
```

```
        {
            sp_vershin = new Dictionary<int, vershina>();
            kol_v_n = 0;
        }
// метод, возвращающий ключ вершины минимальной степени
        public int minDegreeVertex()
        {
            int nom =-1; // номер вершины минимальной степени
            int min ;       // значение минимальной степени
            foreach (KeyValuePair<int, vershina> u in sp_vershin)
            {
                if( nom == -1)
                {
                    nom = u.Key;
                    min = u.Value.degree;
                }
                else if ( u.Value.degree < min)
                {
                    nom = u.Key;
                    min = u.Value.degree;
                }
             }
            return nom;
        }
}
internal class resultSet
{ // результирующее множество :
        public HashSet<int> clique;     // максимальная клика
        public HashSet<int> indepSet; // наибольшее незав. мн- во
        public resultSet()
        {
```

```
            clique = new HashSet<int>();
            indepSet = new HashSet<int>();
        }
    }
// алгоритм аппроксимации независимого множества
class algmApproxIndependentSet        // вершин графа
 {
      resultSet result;
      public algmApproxIndependentSet(Graph g)
      {
         result = Ramsey(g);
         Console.WriteLine("  Independence number = ");
         Console.WriteLine("  {0} ");result.indepSet.Count);
         foreach (int v in result.indepSet)
                    Console.Write("{0}  ", v);
      }
      resultSet Ramsey(Graph g)
      {
         if (g.kol_v_n == 0)
              return new resultSet();
// выберем вершину минимальной степени
         int tek_vna = g.minDegreeVertex();
 // разделим граф на две части: соседей и не соседей tek_vna
         Graph neighbors = buildNeighbGraph(tek_vna, g);
         Graph non_neighbors = buildNonNeighbGraph(tek_vna, g);
 // вызовем алгоритм Рамсей
         resultSet s1 = Ramsey(neighbors);
         resultSet s2 = Ramsey(non_neighbors);
// определим наибольшее мн-во
         s1.clique.Add(tek_vna);
         s2.indepSet.Add(tek_vna);
```

```
        resultSet rez = new resultSet();
        if (s1.clique.Count > s2.clique.Count)
            rez.clique = s1.clique;
         else          rez.clique = s2.clique;
        if (s1.indepSet.Count > s2.indepSet.Count)
            rez.indepSet = s1.indepSet;
        else         rez.indepSet = s2.indepSet;
        return rez;
      }
// метод, который строит граф соседей
    Graph buildNeighbGraph(int v_na, Graph g)
     {
        Graph neighbors = new Graph();
        neighbors.kol_v_n = g.sp_vershin[v_na].sp_reber.Count;
// пройдем по списку смежных вершин
// и занесем их в словарь графа соседей
        foreach (rebro u in g.sp_vershin[v_na].sp_reber)
        {
           vershina v = new vershina();
           neighbors.sp_vershin.Add(u.v_na, v);
        }
 // для каждой вершины графа соседей пройдем по списку
// смежных вершин и если смежная вершина принадлежит графу
// соседей, то добавим в граф соседей соответствующее ребро
       foreach (KeyValuePair<int, vershina> p in neighbors.sp_vershin)
        {
           foreach (rebro u in g.sp_vershin[p.Key].sp_reber)
           {
              if (neighbors.sp_vershin.ContainsKey(u.v_na))
              {
                 rebro r = new rebro { v_na = u.v_na };
```

```
                p.Value.sp_reber.Add(r);
              }
            }
            p.Value.degree = p.Value.sp_reber.Count;
          }
          return neighbors;
        }
// метод, который строит граф несоседей
        Graph buildNonNeighbGraph(int v_na, Graph g)
        {
          Graph non_neighbors = new Graph();
 // пройдем по списку всех вершин графа и если их нет
// в списке смежных вершин текущей, то занесем их в словарь
 // графа несоседей
          foreach (KeyValuePair<int, vershina> p in g.sp_vershin)
          {
            bool flag = true;
            foreach( rebro r in g.sp_vershin[v_na].sp_reber)
            {
              if (r.v_na == p.Key)
                flag = false;
            }
            if (flag && (p.Key!= v_na))
            {
              vershina u = new vershina();
              non_neighbors.sp_vershin.Add(p.Key, u);
            }
          }
          non_neighbors.kol_v_n = non_neighbors.sp_vershin.Count;
// для каждой вершины графа несоседей пройдем по списку
// смежных вершин и если смежная вершина принадлежит графу
```

```
// несоседей, то добавим в граф несоседей соответствующее
// ребро
 foreach (KeyValuePair<int, vershina> p in non_neighbors.sp_vershin)
      {
          foreach (rebro u in g.sp_vershin[p.Key].sp_reber)
          {
             if (non_neighbors.sp_vershin.ContainsKey(u.v_na))
             {
                rebro r = new rebro { v_na = u.v_na };
                p.Value.sp_reber.Add(r);
             }
          }
          p.Value.degree = p.Value.sp_reber.Count;
        }
        return non_neighbors;
      }
}
```

Листинг 12. Алгоритм аппроксимации независимого множества

6.3. Вершинное покрытие

Введем еще одно понятие, связанное с понятием независимости. Будем говорить, что вершина и ребро графа *покрывают* друг друга, если они инцидентны. Таким образом, ребро $e = \{u, v\}$ покрывает вершины u и v, а каждая из этих вершин покрывает ребро e.

Подмножество $V' \subseteq V(G)$ называется *вершинным покрытием* графа G, если каждое ребро из $E(G)$ инцидентно хотя бы одной вершине из V'. Покрытие графа G называется *минимальным*, если оно не содержит покрытия с меньшим числом вершин, и *наименьшим*, если число вершин в нем наименьшее среди всех покрытий графа G.

Задачи о вершинном покрытии и независимом множестве тесно связаны друг с другом. Множество U вершин графа G

является наименьшим (минимальным) покрытием тогда и только тогда, когда $U' = V(G) \setminus U$ – наибольшее независимое множество.

Действительно, каждое ребро графа G по определению инцидентно какой-либо вершине в покрытии U. Поэтому невозможно существование ребра, обе вершины которого принадлежали бы множеству $V(G) \setminus U$. Таким образом, множество $V(G) \setminus U$ является независимым. И минимизация множества U равносильна максимизации множества $V(G) \setminus U$, то есть эти две задачи эквивалентны. Следовательно, задача поиска минимального вершинного покрытия является *NP*-полной. Но с помощью очень простого алгоритма можно быстро найти покрытие, которое, самое большее, вдвое больше оптимального.

Выберем в графе произвольное ребро $\{u, v\}$, добавим обе вершины u, v к вершинному покрытию, удалим все ребра, имеющие общую вершину с этим ребром, и повторяем этот процесс до тех пор, пока в графе не останется больше ребер.

Очевидно, что данный алгоритм строит вершинное покрытие. Теперь докажем, почему любое другое покрытие имеет не более чем в два раза меньшее количество вершин.

Рассмотрим ребра, выбранные алгоритмом. Ни одна пара этих ребер не может иметь общую вершину. Поэтому любое оптимальное покрытие должно содержать по одной вершине на каждое выбранное ребро, следовательно, оно меньше не более чем в два раза.

Возможно более естественным кажется иной эвристический алгоритм, например, при выборе ребра включать только одну вершину из двух, но на графе $K_{1,n}$ при неудачном выборе вершины получим покрытие до n-1 вершин, когда наш алгоритм выдаст две вершины. Или если рассмотреть «жадный» алгоритм, и каждый раз включать в вершинное покрытие вершину наибольшей степени, то на графе $K_{1,n}$ он будет работать идеально, но в случае выбора между одинаковыми или почти одинаковыми вершинами может значительно отклониться от правильного решения. При этом его трудоемкость хуже, так как необходимо каждый раз упорядочивать вершины по степени.

Дополнительным преимуществом простых эвристических алгоритмов является то, что их часто можно модифицировать для получения лучших практических решений. Например, после окончания работы алгоритма провести операцию по удалению ненужных вершин из покрытия, что позволит получить лучший результат.

6.4. Паросочетания

Не менее важным, чем понятие вершинной независимости, является понятие реберной независимости.

Произвольное подмножество попарно несмежных ребер графа называется *паросочетанием* (или независимым множеством ребер).

Паросочетание графа G называется *максимальным*, если оно не содержится в паросочетании с большим числом ребер, и *наибольшим*, если число ребер в нем наибольшее среди всех паросочетаний графа G.

При изучении паросочетаний основное внимание будет уделяться двудольным графам. Для поиска паросочетаний в произвольном графе используются те же идеи, что и в случае двудольных, только реализация их усложняется.

Задача поиска максимального паросочетания в двудольном графе имеет множество практических приложений. В качестве примера можно рассмотреть паросочетание множества компьютеров K и множества задач T, которые должны выполняться одновременно. Наличие в графе ребра $\{u, v\}$ означает, что машина $u \in K$ может выполнять задачу $v \in T$. Максимальное паросочетание обеспечивает максимальную загрузку компьютеров.

Задача построения наибольших паросочетаний в графе изучалась многими исследователями, и для ее решения имеются эффективные алгоритмы, основанные на методе чередующихся цепей Дж. Петерсена.

Пусть M – паросочетание в графе G. Цепь графа G, ребра которой поочередно входят и не входят в M, называется *чередующейся* относительно M. Цепь длины 1, по определению, также чередующаяся. Ребра цепи называются темными или светлыми, если они входят или соответственно не входят в M. Вершины

графа G, инцидентные ребрам из M, называются насыщенными, все другие вершины – ненасыщенными. Очевидно, что если в графе G существует чередующаяся относительно паросочетания M цепь, соединяющая две несовпадающие ненасыщенные вершины, то можно построить в G паросочетание с бóльшим числом ребер, чем в M. В самом деле, в такой цепи число темных ребер на единицу меньше числа светлых.

Удалив из M все темные ребра и присоединив все светлые, получим новое паросочетание, в котором число ребер на единицу больше. По этой причине чередующуюся относительно паросочетания M цепь, соединяющую две различные ненасыщенные вершины, будем называть увеличивающейся относительно M цепью в графе G.

Отсутствие увеличивающихся относительно M цепей необходимо, если паросочетание M наибольшее. Поэтому разработка эффективного алгоритма сводится к построению процедуры, которая быстро находит увеличивающую цепь в графе либо выявляет ее отсутствие.

Пусть $G = (X, Y, E)$ – двудольный граф и M – паросочетание в этом графе. Поставим в соответствие графу G и паросочетанию M вспомогательный ориентированный двудольный граф $\vec{G}$, такой что ребра графа G, входящие в паросочетание M, соответствуют в $\vec{G}$ дугам, направленным от Y к X, а все остальные ребра соответствуют дугам, направленным от X к Y.

Обозначим через X_M и Y_M множества ненасыщенных вершин, входящих соответственно в X и Y. Очевидно, что в графе G увеличивающаяся относительно паросочетания M цепь существует тогда и только тогда, когда в графе $\vec{G}$ существует (s, t)-путь, у которого $s \in X_M$, $t \in Y_M$.

Пусть $\vec{P}$ – (s, t)-путь в графе $\vec{G}$, $s \in X_M$, $t \in Y_M$, P – соответствующая увеличивающаяся цепь в графе G и M_1 – паросочетание, полученное изменением M вдоль цепи P. Тогда вспомогательный граф $\vec{G}_1$ для графа G и паросочетания M_1 можно получить из графа $\vec{G}$ заменой каждой дуги пути $\vec{P}$ на обратную. Эта операция вместе с поиском пути $\vec{P}$ составляет итерацию приводимого алгоритма.

Предполагается, что граф G задан списками смежности.

Алгоритм построения наибольшего паросочетания в двудольном графе.

1. Построить какое-либо максимальное паросочетание M в графе G.

2. По графу G и паросочетанию M построить граф $\vec{G}$.

3. Выполнить в графе $\vec{G}$ поиск в ширину из множества X_M вершин, не насыщенных текущим паросочетанием.

4. Если в результате поиска в ширину ни одна из вершин множества Y_M не получила метки, то перейти к п. 5. Иначе – перейти к п. 6.

5. Наибольшее паросочетание M^* равно множеству всех дуг, выходящих из множества Y. Конец.

6. Пусть $\vec{P}$ – (s, t)-путь в графе $\vec{G}$ такой, что $s \in X_M$, $t \in Y_M$. Изменить граф $\vec{G}$, заменив в нем каждую дугу (a, b) пути $\vec{P}$ на дугу (b, a). Перейти к п. 2.

Вместе с наибольшим паросочетанием алгоритм фактически находит и наименьшее вершинное покрытие в графе G. В результате последнего выполнения п. 3 алгоритма будет установлено отсутствие пути из X_M в Y_M в графе G. Следовательно, только часть вершин этого графа будет иметь метки после окончания поиска в ширину из X_M. Обозначим через X' и Y' соответственно множества непомеченных вершин доли X и помеченных вершин доли Y. Положим $Z = X' \cup Y'$. Множество Z является наименьшим вершинным покрытием графа G.

Трудоемкость алгоритма построения наибольшего паросочетания составляет $O(m \cdot \min\{|X|, |Y|\})$. Действительно, на каждой итерации алгоритма, кроме последней, размер множеств X_M и Y_M уменьшается на единицу. Поэтому общее число итераций не больше $\min\{|X|, |Y|\}$. Поиск в ширину (п. 3 алгоритма) выполняется за $O(m)$.

Упражнения

1. Приведите пример графа, на котором алгоритм построения наибольшего независимого множества путем выбора вершин минимальной степени дает не лучший результат.

2. Подмножество V' вершин графа G называется *доминирующим,* если каждая вершина из $V(G) \setminus V'$ смежна с некоторой вершиной из V'. Докажите, что независимое множество является максимальным (не обязательно наибольшим) тогда и только тогда, когда оно доминирующее.

Приведите пример графа, в котором доминирующее множество не является независимым.

3. Верно ли, что любое паросочетание графа содержится в наибольшем паросочетании?

4. Напишите реализацию алгоритма поиска наибольшего паросочетания в двудольном графе.

Список литературы

1. Кормен, Т. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн; пер. с англ. – 2-е изд. – М.: Вильямс, 2011. – 1 296 с.

2. Кристофидес, Н. Теория графов: Алгоритмический подход / Н. Кристофидес. – М.: Мир, 1978. – 434 с.

3. Седжвик, Р. Фундаментальные алгоритмы на C++: Алгоритмы на графах / Р. Седжвик; пер. с англ. – СПб.: Диасофт, 2002. – 496 с.

4. Скиена, С. Алгоритмы. Руководство по разработке / С. Скиена; пер. с англ. – 2-е изд. – СПб.: БХВ-Петербург, 2011.– 720 с.

5. Троелсен, Э. C# и платформа .NET 3.0, специальное издание / Э. Троелсен. – СПб.: Питер, 2008. – 1 456 с.

6. Boppana, R. Approximating maximum independent sets by excluding subgraphs / R. Boppana, M. Halldorsson // Bit 32. – 1992. – № 2 (June).

7. Erdős, P. A combinatorial problem in geometry / P. Erdős, G. Szekerres // Compositio Math. – 1935. – Vol. 2. – P. 463–470.

Оглавление

Учебное издание

Дольников Владимир Леонидович
Якимова Ольга Павловна

Основные алгоритмы на графах

Текст лекций

Редактор, корректор М. В. Никулина
Верстка Е. Л. Шелехова

Подписано в печать 24.10.11. Формат 60×84 $^{1}/_{16}$.
Бум. офсетная. Гарнитура "Times New Roman".
Усл. печ. л. 4,65. Уч.-изд. л. 3,03.
Тираж 50 экз. Заказ .

Оригинал-макет подготовлен
в редакционно-издательском отделе Ярославского
государственного университета им. П. Г. Демидова.

Отпечатано на ризографе.

Ярославский государственный университет
им. П. Г. Демидова.
150000, Ярославль, ул. Советская, 14.

В. Л. Дольников
О. П. Якимова

Основные алгоритмы на графах